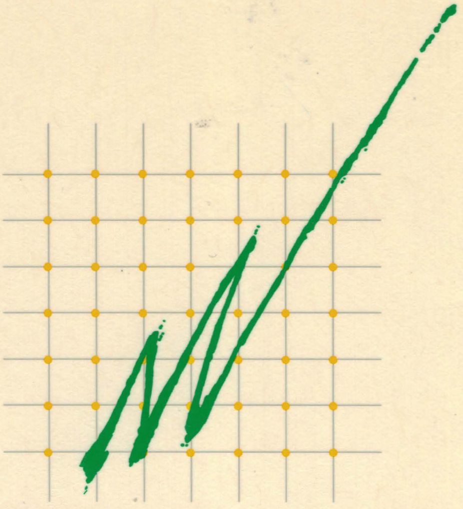
MC68020UM/AD
REV 2



MC68020

32-BIT MICROPROCESSOR USER'S MANUAL
THIRD EDITION

MOTOROLA

MC68020 USER'S MANUAL

MOTOROLA

PRENTICE
HALL

Introduction 1

Data Organization and Addressing Capabilities 2

Instruction Set 3

Processing States 4

Signal Description 5

On-Chip Cache Memory 6

Bus Operation 7

Exception Processing 8

Coprocessor Interface Description 9

Instruction Execution Timing 10

Applications Information 11

Electrical Specifications 12

Ordering Information and Mechanical Data 13

Index I

MOTOROLA

MC68020

32-BIT MICROPROCESSOR USER'S MANUAL

Third Edition

PRENTICE HALL, Englewood Cliffs, New Jersey 07632

1989, 1988 by Motorola Inc.

All rights reserved. No part of this book may be
reproduced, in any form or by any means,
without permission in writing from the publisher.

This document contains information on a new product. Specifications and information herein are subject to change without notice. Motorola reserves the right to make changes to any products herein to improve functioning or design. Although the information in this document has been carefully reviewed and is believed to be reliable, Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others.

Motorola, Inc. general policy does not recommend the use of its components in life support applications wherein a failure or malfunction of the component may directly threaten life or injury. Per Motorola Terms and Conditions of Sale, the user of Motorola components in life support applications assumes all risk of such use and indemnifies Motorola against all damages.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-567017-9

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

PREFACE

The *MC68020 User's Manual* describes the capabilities, operation, and programming of the MC68020 32-bit second-generation enhanced microprocessor. The manual consists of the following sections.

Section 1. Introduction
Section 2. Data Organization and Addressing Capabilities
Section 3. Instruction Set
Section 4. Processing States
Section 5. Signal Description
Section 6. On-Chip Cache Memory
Section 7. Bus Operation
Section 8. Exception Processing
Section 9. Coprocessor Interface Description
Section 10. Instruction Execution Timing
Section 11. Applications Information
Section 12. Electrical Specifications
Section 13. Ordering Information and Mechanical Data

NOTE

In this manual, assertion and negation are used to specify forcing a signal to a particular state. In particular, **assertion** and **assert** refer to a signal that is active or true; **negation** and **negate** indicate a signal that is inactive or false. These terms are used independently of the voltage level (high or low) that they represent.

The audience of this manual includes system designers, systems programmers, and applications programmers. Systems designers need some knowledge of all sections, with particular emphasis on Sections 1, 5, 6, 7, 12, and 13. Designers who implement a coprocessor for their system also need a thorough knowledge of Section 10. Systems programmers should become familiar with Sections 1, 2, 3, 4, 6, 8, and 10. Applications programmers can find most of the information they need in Sections 1, 2, 3, 4, 10, and 11.

From a different viewpoint, the audience for this book consists of users of other M68000 Family members, and those who are not familiar with these microprocessors. Users of the other family members can find references to similarities to and differences from the other MOTOROLA microprocessors throughout the manual. However, Section 1 specifically identifies the MC68020 within the rest of the family, and contrast its differences.

TABLE OF CONTENTS

Paragraph Number	Title	Page Number

Section 1
Introduction

1.1 Features 1-1
1.2 Programming Model 1-3
1.3 Data Types and Addressing Modes 1-5
1.4 Instruction Set Overview 1-6
1.5 Virtual Memory and Virtual Machine Concepts 1-6
1.5.1 Virtual Memory 1-7
1.5.2 Virtual Machine 1-9
1.6 Pipelined Architecture 1-9
1.7 The Cache Memory 1-10

Section 2
Data Organization and Addressing Capabilities

2.1 Instruction Operands 2-1
2.2 Organization of Data in Registers 2-1
2.2.1 Data Registers 2-2
2.2.2 Address Registers 2-3
2.2.3 Control Registers 2-4
2.3 Organization of Data in Memory 2-4
2.4 Addressing Modes 2-5
2.4.1 Data Register Direct Mode 2-8
2.4.2 Address Register Direct Mode 2-8
2.4.3 Address Register Indirect Mode 2-8
2.4.4 Address Register Indirect with Postincrement Mode 2-8
2.4.5 Address Register Indirect with Predecrement Mode 2-9
2.4.6 Address Register Indirect with Displacement Mode 2-9
2.4.7 Address Register Indirect with Index (8-Bit Displacement) Mode 2-10
2.4.8 Address Register Indirect with Index (Base Displacement) Mode 2-10
2.4.9 Memory Indirect Postindexed Mode 2-11
2.4.10 Memory Indirect Preindexed Mode 2-11
2.4.11 Program Counter Indirect with Displacement Mode 2-12
2.4.12 Program Counter Indirect with Index (8-Bit Displacement) 2-13
2.4.13 Program Counter Indirect with Index (Base Displacement) Mode 2-13
2.4.14 Program Counter Memory Indirect Postindexed Mode 2-14
2.4.15 Program Counter Memory Indirect Preindexed Mode 2-15
2.4.16 Absolute Short Address Mode 2-16
2.4.17 Absolute Long Address Mode 2-16
2.4.18 Immediate Data 2-16

TABLE OF CONTENTS (Continued)

Paragraph Number	Title	Page Number
2.5	Effective Address Encoding Summary	2-17
2.6	Programmer's View of Addressing Modes	2-19
2.6.1	Addressing Capabilities	2-19
2.6.2	General Addressing Mode Summary	2-23
2.7	M68000 Family Addressing Compatibility	2-27
2.8	Other Data Structures	2-28
2.8.1	System Stack	2-29
2.8.2	User Program Stacks	2-29
2.8.3	Queues	2-30

Section 3
Instruction Set

Paragraph Number	Title	Page Number
3.1	Instruction Format	3-1
3.2	Instruction Summary	3-2
3.2.1	Data Movement Instructions	3-3
3.2.2	Integer Arithmetic Instructions	3-3
3.2.3	Logical Instructions	3-4
3.2.4	Shift and Rotate Instructions	3-4
3.2.5	Bit Manipulation Instructions	3-7
3.2.6	Bit Field Instructions	3-7
3.2.7	Binary Coded Decimal Instructions	3-8
3.2.8	Program Control Instructions	3-8
3.2.9	System Control Instructions	3-9
3.2.10	Multiprocessor Instructions	3-10
3.3	Instruction Set Details	3-10
3.3.1	Notation and Format	3-11
3.3.2	Condition Code Register	3-12
3.3.3	Instruction Descriptions	3-13
3.4	Using the CAS and CAS2 Instructions	3-194
3.5	Nested Subroutine Calls	3-197
3.6	Bit Field Instructions	3-198
3.7	Pipeline Synchronization with the NOP Instruction	3-199
3.8	Condition Codes	3-199
3.8.1	Condition Code Computation	3-200
3.8.2	Condition Tests	3-201
3.9	Instruction Format Summary	3-201

Section 4
Processing States

Paragraph Number	Title	Page Number
4.1	Privilege Levels	4-1
4.1.1	Supervisor Privilege Level	4-2
4.1.2	User Privilege Level	4-3
4.1.3	Changing Privilege Level	4-3

TABLE OF CONTENTS (Continued)

Paragraph Number — Title — Page Number

4.2 Address Space Types 4-4
4.3 Exception Processing 4-4
4.3.1 Exception Vectors 4-5
4.3.2 Exception Stack Frame 4-5

Section 5
Signal Description

5.1 Signal Index 5-2
5.2 Function Code Signals (FC2–FC0) 5-3
5.3 Address Bus (A31–A0) 5-3
5.4 Data Bus (D31–D0) 5-3
5.5 Transfer Size Signals (SIZ0, SIZ1) 5-3
5.6 Bus Control Signals 5-3
5.6.1 Operand Cycle Start ($\overline{\text{OCS}}$) 5-3
5.6.2 External Cycle Start ($\overline{\text{ECS}}$) 5-3
5.6.3 Read/Write (R/$\overline{\text{W}}$) 5-4
5.6.4 Read-Modify-Write Cycle ($\overline{\text{RMC}}$) 5-4
5.6.5 Address Strobe ($\overline{\text{AS}}$) 5-4
5.6.6 Data Strobe ($\overline{\text{DS}}$) 5-4
5.6.7 Data Buffer Enable ($\overline{\text{DBEN}}$) 5-4
5.6.8 Data Transfer and Size Acknowledge ($\overline{\text{DSACK1}}$, $\overline{\text{DSACK0}}$) 5-4
5.7 Interrupt Control Signals 5-5
5.7.1 Interrupt Priority Level Signals 5-5
5.7.2 Interrupt Pending ($\overline{\text{IPEND}}$) 5-5
5.7.3 Autovector ($\overline{\text{AVEC}}$) 5-5
5.8 Bus Arbitration Control Signals 5-5
5.8.1 Bus Request ($\overline{\text{BR}}$) 5-5
5.8.2 Bus Grant ($\overline{\text{BG}}$) 5-5
5.8.3 Bus Grant Acknowledge ($\overline{\text{BGACK}}$) 5-6
5.9 Bus Exception Control Signals 5-6
5.9.1 Reset ($\overline{\text{RESET}}$) 5-6
5.9.2 Halt ($\overline{\text{HALT}}$) 5-6
5.9.3 Bus Error ($\overline{\text{BERR}}$) 5-6
5.10 Emulator Support Signals 5-6
5.10.1 Cache Disable ($\overline{\text{CDIS}}$) 5-6
5.11 Clock (CLK) 5-7
5.12 Power Supply Connections 5-7
5.13 Signal Summary 5-7

Section 6
On-Chip Cache Memories

6.1 On-Chip Cache Organization and Operation 6-1
6.2 Cache Reset 6-2

TABLE OF CONTENTS (Continued)

Paragraph Number — Title — Page Number

6.3 Cache Control 6-2
6.3.1 Cache Control Register 6-2
6.3.2 Clear Cache 6-3
6.3.3 Clear Entry In Cache 6-3
6.3.4 Freeze Cache 6-3
6.3.5 Enable Cache 6-3
6.3.6 Cache Address Register 6-4

Section 7
Bus Operation

7.1 Bus Transfer Signals 7-1
7.1.1 Bus Control Signals 7-2
7.1.2 Address Bus 7-3
7.1.3 Address Strobe 7-3
7.1.4 Data Bus 7-3
7.1.5 Data Strobe 7-4
7.1.6 Data Buffer Enable 7-4
7.1.7 Bus Cycle Termination Signals 7-4
7.2 Data Transfer Mechanism 7-5
7.2.1 Dynamic Bus Sizing 7-5
7.2.2 Misaligned Operands 7-9
7.2.3 Effects of Dynamic Bus Sizing and Operand Misalignment 7-11
7.2.4 Address, Size, and Data Bus Relationships 7-13
7.2.5 Cache Interactions 7-19
7.2.6 Bus Operation 7-21
7.2.7 Synchronous Operation with $\overline{\text{DSACKx}}$ 7-21
7.3 Data Transfer Cycles 7-22
7.3.1 Read Cycle 7-23
7.3.2 Write Cycle 7-28
7.3.3 Read-Modify-Write Cycle 7-34
7.4 CPU Space Cycles 7-38
7.4.1 Interrupt Acknowledge Bus Cycles 7-38
7.4.1.1 Interrupt Acknowledge Cycle — Terminated Normally 7-39
7.4.1.2 Autovector Interrupt Acknowledge Cycle 7-39
7.4.1.3 Spurious Interrupt Cycle 7-40
7.4.2 Breakpoint Acknowledge Cycle 7-40
7.4.3 Coprocessor Communication Cycles 7-43
7.5 Bus Exception Control Cycles 7-45
7.5.1 Bus Errors 7-47
7.5.2 Retry Operation 7-48
7.5.3 Halt Operation 7-50
7.5.4 Double Bus Fault 7-53
7.6 Bus Synchronization 7-53

TABLE OF CONTENTS (Continued)

Paragraph Number | Title | Page Number

7.7 Bus Arbitration 7-54
7.7.1 Bus Request 7-55
7.7.2 Bus Grant 7-57
7.7.3 Bus Grant Acknowledge 7-57
7.7.4 Bus Arbitration Control 7-57
7.8 Reset Operation 7-60

Section 8
Exception Processing

8.1 Exception Processing Sequence 8-1
8.1.1 Reset Exception 8-2
8.1.2 Bus Error Exception 8-4
8.1.3 Address Error Exception 8-5
8.1.4 Instruction Trap Exception 8-5
8.1.5 Illegal Instruction and Unimplemented Instruction Exceptions 8-6
8.1.6 Privilege Violation Exception 8-7
8.1.7 Trace Exception 8-7
8.1.8 Format Error Exception 8-9
8.1.9 Interrupt Exceptions 8-9
8.1.10 Breakpoint Instruction Exception 8-14
8.1.11 Multiple Exception 8-15
8.1.12 Return from Exceptions 8-16
8.2 Bus Fault Recovery 8-18
8.2.1 Special Status Word 8-19
8.2.2 Using Software to Complete the Bus Cycles 8-20
8.2.3 Completing the Bus Cycles with RTE 8-21
8.3 Coprocessor Considerations 8-22
8.4 Exception Stack Frame Formats 8-22

Section 9
Coprocessor Interface Description

9.1 Introduction 9-1
9.1.1 Interface Features 9-2
9.1.2 Concurrent Operation Support 9-2
9.1.3 Coprocessor Instruction Format 9-3
9.1.4 Coprocessor System Interface 9-4
9.1.4.1 Coprocessor Classification 9-4
9.1.4.2 Processor-Coprocessor Interface 9-5
9.1.4.3 Coprocessor Interface Register Selection 9-6
9.2 Coprocessor Instruction Types 9-7
9.2.1 Coprocessor General Instructions 9-8
9.2.1.1 Format 9-8
9.2.1.2 Protocol 9-9

TABLE OF CONTENTS (Continued)

Paragraph Number	Title	Page Number
9.2.2	Coprocessor Conditional Instructions	9-10
9.2.2.1	Branch On Coprocessor Condition Instruction	9-10
9.2.2.1.1	Format	9-10
9.2.2.1.2	Protocol	9-12
9.2.2.2	Set On Coprocessor Condition Instruction	9-12
9.2.2.2.1	Format	9-12
9.2.2.2.2	Protocol	9-13
9.2.2.3	Test Coprocessor Condition, Decrement and Branch Instruction	9-13
9.2.2.3.1	Format	9-13
9.2.2.3.2	Protocol	9-14
9.2.2.4	Trap On Coprocessor Condition	9-14
9.2.2.4.1	Format	9-14
9.2.2.4.2	Protocol	9-15
9.2.3	Coprocessor Save and Restore Instructions	9-16
9.2.3.1	Coprocessor Internal State Frames	9-16
9.2.3.2	Coprocessor Format Words	9-17
9.2.3.2.1	Empty/Reset Format Word	9-18
9.2.3.2.2	Not Ready Format Word	9-18
9.2.3.2.3	Invalid Format Word	9-18
9.2.3.2.4	Valid Format Word	9-19
9.2.3.3	Coprocessor Context Save Instruction	9-19
9.2.3.3.1	Format	9-19
9.2.3.3.2	Protocol	9-20
9.2.3.4	Coprocessor Context Restore Instruction	9-21
9.2.3.4.1	Format	9-21
9.2.3.4.2	Protocol	9-22
9.3	Coprocessor Interface Register Set	9-23
9.3.1	Response CIR	9-23
9.3.2	Control CIR	9-23
9.3.3	Save CIR	9-24
9.3.4	Restore CIR	9-24
9.3.5	Operation Word CIR	9-24
9.3.6	Command CIR	9-24
9.3.7	Condition CIR	9-25
9.3.8	Operand CIR	9-25
9.3.9	Register Select CIR	9-26
9.3.10	Instruction Address CIR	9-26
9.3.11	Operand Address CIR	9-26
9.4	Coprocessor Response Primitives	9-26
9.4.1	ScanPC	9-26
9.4.2	Coprocessor Response Primitive General Format	9-27
9.4.3	Busy Primitive	9-28
9.4.4	Null Primitive	9-29
9.4.5	Supervisor Check Primitive	9-30

TABLE OF CONTENTS (Continued)

Paragraph Number — Title — Page Number

9.4.6 Transfer Operation Word Primitive 9-31
9.4.7 Transfer from Instruction Stream Primitive 9-32
9.4.8 Evaluate and Transfer Effective Address Primitive 9-33
9.4.9 Evaluate Effective Address and Transfer Data Primitive 9-33
9.4.10 Write to Previously Evaluated Effective Address Primitive 9-35
9.4.11 Take Address and Transfer Data Primitive 9-37
9.4.12 Transfer to/from Top of Stack Primitive 9-37
9.4.13 Transfer Single Main Processor Register Primitive 9-38
9.4.14 Transfer Main Processor Control Register Primitive 9-38
9.4.15 Transfer Multiple Main Processor Registers Primitive 9-39
9.4.16 Transfer Multiple Coprocessor Registers Primitive 9-40
9.4.17 Transfer Status Register and ScanPC Primitive 9-41
9.4.18 Take Pre-Instruction Exception Primitive 9-43
9.4.19 Take Mid-Instruction Exception Primitive 9-44
9.4.20 Take Post-Instruction Exception Primitive 9-45
9.5 Exceptions 9-47
9.5.1 Coprocessor Detected Exceptions 9-47
9.5.1.1 Coprocessor Detected Protocol Violations 9-47
9.5.1.2 Coprocessor Detected Illegal Command or Condition Words ... 9-48
9.5.1.3 Coprocessor Data Processing Exceptions 9-48
9.5.1.4 Coprocessor System Related Exceptions 9-49
9.5.1.5 Format Errors 9-49
9.5.2 Main Processor Detected Exceptions 9-49
9.5.2.1 Protocol Violations 9-49
9.5.2.2 F-Line Emulator Exceptions 9-51
9.5.2.3 Privilege Violations 9-52
9.5.2.4 cpTRAPcc Instruction Traps 9-52
9.5.2.5 Trace Exceptions 9-52
9.5.2.6 Interrupts 9-53
9.5.2.7 Main Processor Detected Format Errors 9-53
9.5.2.8 Address and Bus Errors 9-54
9.5.3 Coprocessor Reset 9-54
9.6 Coprocessor Summary 9-54

Section 10
Instruction Execution Timing

10.1 Timing Estimation Factors 10-1
10.1.1 Instruction Cache and Prefetch 10-1
10.1.2 Operand Misalignment 10-2
10.1.3 Bus/Sequencer Concurrency 10-2
10.1.4 Instruction Execution Overlap 10-2
10.1.5 Instruction Stream Timing Examples 10-3

TABLE OF CONTENTS (Continued)

Paragraph Number	Title	Page Number
10.2	Instruction Timing Tables	10-8
10.2.1	Fetch Effective Address	10-12
10.2.2	Fetch Immediate Effective Address	10-13
10.2.3	Calculate Effective Address	10-15
10.2.4	Calculate Immediate Effective Address	10-16
10.2.5	Jump Effective Address	10-18
10.2.6	MOVE Instruction	10-19
10.2.7	Special Purpose MOVE Instruction	10-28
10.2.8	Arithmetic/Logical Instructions	10-29
10.2.9	Immediate Arithmetic/Logical Instructions	10-30
10.2.10	Binary Coded Decimal Operations	10-31
10.2.11	Single Operand Instructions	10-32
10.2.12	Shift/Rotate Instructions	10-33
10.2.13	Bit Manipulation Instructions	10-34
10.2.14	Bit Field Manipulation Instructions	10-35
10.2.15	Conditional Branch Instructions	10-36
10.2.16	Control Instructions	10-37
10.2.17	Exception-Related Instructions	10-38
10.2.18	Save and Restore Operations	10-39

Section 11
Applications Information

11.1	Module Support	11-1
11.1.1	Module Descriptor	11-1
11.1.2	Module Stack Frame	11-3
11.2	Access Levels	11-3
11.2.1	Module Call	11-5
11.2.2	Module Return	11-5
11.3	Floating-Point Units	11-6

Section 12
Electrical Characteristics

12.1	Maximum Ratings	12-1
12.2	Thermal Characteristics — PGA Package	12-1
12.3	Power Considerations	12-1
12.4	DC Electrical Characteristics	12-2
12.5	AC Electrical Specifications — Clock Input	12-3
12.6	AC Electrical Specifications — Read and Write Cycles	12-4
12.7	AC Electrical Specifications Definitions	12-10

TABLE OF CONTENTS (Concluded)

Paragraph Number	Title	Page Number

Section 13
Ordering Information and Mechanical Data

13.1 Standard MC68020 Ordering Information 13-1
13.2 Pin Assignments and Package Dimensions 13-2
13.2.1 RC, RL Suffix — Pin Assignment 13-2
13.2.2 RC Suffix — Package Dimensions 12-3
13.2.3 RL Suffix — Package Dimensions 13-4
13.2.4 FE Suffix — Pin Assignment 13-5
13.2.5 FE Suffix — Package Dimensions 13-6

INDEX

LIST OF ILLUSTRATIONS

Figure Number	Title	Page Number
1-1	MC68020 Block Diagram	1-2
1-2	User Programming Model	1-3
1-3	Supervisor Programming Model Supplement	1-4
1-4	Status Register	1-5
1-5	MC68020 Pipeline	1-10
2-1	Memory Operand Addressing	2-5
2-2	Memory Data Organization	2-6
2-3	Single-Effective-Address Instruction Operation Word	2-7
2-4	Using SIZE in the Index Selection	2-20
2-5	Using Absolute Address with Indexes	2-20
2-6	Addressing Array Items	2-21
2-7	Using Indirect Absolute Memory Addressing	2-22
2-8	Accessing an Item in a Structure Using Pointer	2-22
2-9	Indirect Addressing, Suppressed Index Register	2-23
2-10	Preindexed Indirect Addressing	2-24
2-11	Postindexed Indirect Addressing	2-24
2-12	Preindexed Indirect with Outer Displacement	2-25
2-13	Postindexed Indirect Addressing with Outer Displacement	2-25
2-14	M68000 Family Address Extension Words	2-28
3-1	Instruction Word General Format	3-1
3-2	Instruction Description Format	3-16
3-3	Linked List Insertion	3-195
3-4	Linked List Deletion	3-196
3-5	Doubly Linked List Insertion	3-197
3-6	Doubly Linked List Deletion	3-198
4-1	General Exception Stack Frame	4-6
5-1	Functional Signal Groups	5-1
6-1	MC68020 On-Chip Cache Organization	6-2
6-2	Cache Control Register	6-3
6-3	Cache Address Register	6-3
7-1	Relationship Between External and Internal Signals	7-2
7-2	Input Sample Window	7-2
7-3	Internal Operand Representation	7-6
7-4	MC68020 Interface to Various Port Sizes	7-7

LIST OF ILLUSTRATIONS (Continued)

Figure Number	Title	Page Number
7-5	Example of Long-Word Transfer to Word Port	7-8
7-6	Long-Word Operand Write Timing (16-Bit Data Port)	7-10
7-7	Example of Word Transfer to Byte Port	7-11
7-8	Word Operand Write Timing (8-Bit Data Port)	7-12
7-9	Misaligned Long-Word Transfer to Word Port Example	7-13
7-10	Misaligned Long-Word Transfer to Word Port	7-14
7-11	Misaligned Long-Word Transfer from Word Port Example	7-15
7-12	Misaligned Word Transfer to Word Port Example	7-15
7-13	Misaligned Word Transfer to Word Port	7-16
7-14	Example of Misaligned Word Transfer from Word Bus	7-17
7-15	Misaligned Long-Word Transfer to Long-Word Port	7-17
7-16	Misaligned Write Cycles to Long-Word Port	7-18
7-17	Misaligned Long-Word Transfer from Long-Word Bus	7-19
7-18	Byte Data Select Generation for 16- and 32-Bit Ports	7-20
7-19	Asynchronous Long-Word Read Cycle Flowchart	7-23
7-20	Byte-Read Cycle Flowchart	7-24
7-21	Byte and Word-Read Cycles — 32-Bit Port	7-25
7-22	Long-Word Read — 8-Bit Port	7-26
7-23	Long-Word Read — 16-Bit Port and 32-Bit Port	7-27
7-24	Write Cycle Flowchart	7-28
7-25	Read-Write-Read Cycles — 32-Bit Port	7-29
7-26	Byte and Word-Write Cycles — 32-Bit Port	7-30
7-27	Long-Word Operand Write — 8-Bit Port	7-31
7-28	Long-Word Operand Write — 16-Bit Port	7-32
7-29	Read-Modify-Write Cycle Flowchart	7-35
7-30	Byte Read-Modify-Write Cycle — 32-Bit Port (TAS Instruction)	7-36
7-31	MC68020 CPU-Space Address Encoding	7-38
7-32	Interrupt Acknowledge Cycle Flowchart	7-40
7-33	Interrupt Acknowledge Cycle Timing	7-41
7-34	Autovector Operation Timing	7-42
7-35	Breakpoint Operation Flow	7-43
7-36	Breakpoint Acknowledge Cycle Timing	7-44
7-37	Breakpoint Acknowledge Cycle Timing (Exception Signaled)	7-45
7-38	Bus Error without $\overline{\text{DSACK}}$	7-49
7-39	Late Bus Error with $\overline{\text{DSACK}}$	7-50
7-40	Late Retry	7-51
7-41	Halt Operation Timing	7-52
7-42	Bus Arbitration Flowchart for Single Request	7-55
7-43	Bus Arbitration Operation Timing	7-56
7-44	Bus Arbitration State Diagram	7-58
7-45	Bus Arbitration Operation (Bus Inactive)	7-59
7-46	Initial Reset Operation Timing	7-60
7-47	Processor Generated Reset Operation	7-61

LIST OF ILLUSTRATIONS (Continued)

Figure Number	Title	Page Number
8-1	Reset Operation Flowchart	8-4
8-2	Interrupt Pending Procedure	8-10
8-3	Interrupt Recognition Examples	8-12
8-4	Assertion of $\overline{\text{IPEND}}$	8-12
8-5	Interrupt Exception Processing Flowchart	8-13
8-6	Breakpoint Instruction Flowchart	8-15
8-7	RTE Instruction for Throwaway Four-Word Frames	8-17
8-8	Special Status Word (SSW)	8-19
9-1	F-Line Coprocessor Instruction Operation Word	9-3
9-2	Asynchronous Non-DMA M68000 Coprocessor Interface Signal Usage	9-5
9-3	MC68020 CPU Space Address Encodings	9-6
9-4	Coprocessor Address Map in MC68020 CPU Space	9-7
9-5	Coprocessor Interface Register Set Map	9-7
9-6	Coprocessor General Instruction Format (cpGEN)	9-8
9-7	Coprocessor Interface Protocol for General Category Instructions	9-9
9-8	Coprocessor Interface Protocol for Conditional Category Instructions	9-11
9-9	Branch on Coprocessor Condition Instruction (cpBcc.W)	9-11
9-10	Branch on Coprocessor Condition Instruction (cpBcc.L)	9-11
9-11	Set On Coprocessor Condition (cpScc)	9-12
9-12	Test Coprocessor Condition, Decrement and Branch Instruction Format (cpDBcc)	9-13
9-13	Trap On Coprocessor Condition (cpTRAPcc)	9-15
9-14	Coprocessor State Frame Format in Memory	9-16
9-15	Coprocessor Context Save Instruction Format (cpSAVE)	9-19
9-16	Coprocessor Context Save Instruction Protocol	9-20
9-17	Coprocesor Context Restore Instruction Format (cpRESTORE)	9-21
9-18	Coprocessor Context Restore Instruction Protocol	9-22
9-19	Control CIR Format	9-24
9-20	Condition CIR Format	9-25
9-21	Operand Alignment for Operand CIR Accesses	9-25
9-22	Coprocessor Response Primitive Format	9-27
9-23	Busy Primitive Format	9-28
9-24	Null Primitive Format	9-29
9-25	Supervisor Check Primitive Format	9-31
9-26	Transfer Operation Word Primitive Format	9-32
9-27	Transfer from Instruction Stream Primitive Format	9-32
9-28	Evaluate and Transfer Effective Address Primitive Format	9-33
9-29	Evaluate Effective Address and Transfer Data Primitive Format	9-33
9-30	Write to Previously Evaluated Effective Address Primitive Format	9-35
9-31	Take Address and Transfer Data Primitive Format	9-37
9-32	Transfer To/From Top of Stack Primitive Format	9-37
9-33	Transfer Single Main Processor Register Primitive Format	9-38

LIST OF ILLUSTRATIONS (Concluded)

Figure Number	Title	Page Number
9-34	Transfer Main Processor Control Register Primitive Format	9-39
9-35	Transfer Multiple Main Processor Registers Primitive Format	9-39
9-36	Register Select Mask Format	9-40
9-37	Transfer Multiple Coprocessor Registers Primitive Format	9-40
9-38	Operand Format in Memory for Transfer to −(An)	9-42
9-39	Transfer Status Register and ScanPC Primitive Format	9-42
9-40	Take Pre-Instruction Exception Primitive Format	9-43
9-41	MC68020 Pre-Instruction Stack Frame	9-43
9-42	Take Mid-Instruction Exception Primitive Format	9-44
9-43	MC68020 Mid-Instruction Stack Frame	9-45
9-44	Take Post-Instruction Exception Primitive Format	9-46
9-45	MC68020 Post-Instrution Stack Frame	9-46
10-1	Concurrent Instruction Execution	10-2
10-2	Instruction Execution for Instruction Timing Purposes	10-3
10-3	Processor Activity for Example 1	10-4
10-4	Processor Activity for Example 2	10-5
10-5	Processor Activity for Example 3	10-6
10-6	Processor Activity for Example 4	10-7
11-1	Module Descriptor Format	11-1
11-2	Module Entry Word	11-2
11-3	Module Call Stack Frame	11-3
11-4	Access Level Control Bus Registers	11-4
11-5	32-Bit Data Bus Coprocessor Connection	11-7
11-6	Chip Select Generatio PAL	11-8
11-7	PAL Equations	11-8
11-8	Bus Cycle Timing Diagram	11-9
12-1	Clock Input Timing Diagram	12-3
12-2	Read Cycle Timing Diagram	12-7
12-3	Write Cycle Timing Diagram	12-8
12-4	Bus Arbitration Timing Diagram	12-9
12-5	Drive Levels and Test Points for AC Sepcifications	12-11

LIST OF TABLES

Table Number	Title	Page Number
1-1	MC68030 Addressing Modes	1-7
1-2	Instruction Set	1-8
2-1	Effective Address Specification Formats	2-17
2-2	IS-I/IS Memory Indirection Encodings	2-18
2-3	Effective Addressing Mode Categories	2-19
3-1	Data Movement Operations	3-4
3-2	Integer Arithmetic Operations	3-5
3-3	Logical Operations	3-5
3-4	Shift and Rotate Operations	3-6
3-5	Bit Manipulation Operations	3-7
3-6	Bit Field Operations	3-7
3-7	Binary Coded Decimal Operations	3-7
3-8	Program Control Operations	3-8
3-9	System Control Operations	3-9
3-10	Multiprocessor Operations	3-11
3-11	Condition Code Computations	3-14
3-12	Condition Code Computations	3-200
3-13	Conditional Tests	3-202
3-14	Operation Code Map	3-202
4-1	Address Space Encodings	4-4
5-1	Signal Index	5-2
5-2	Signal Summary	5-8
7-1	$\overline{\text{DSACK}}$ Codes and Results	7-4
7-2	Size Signal Encoding	7-6
7-3	Address Offset Encodings	7-6
7-4	Data Bus Requirements for Read Cycles	7-8
7-5	MC68020 Internal to External Data Bus Multiplexer — Write Cycles	7-9
7-6	Memory Alignment and Port Size Influence on Write Bus Cycles	7-13
7-7	Data Bus Write Enable Signals for Byte, Word, and Long-Word Ports	7-19
7-8	$\overline{\text{DSACK}}$, $\overline{\text{BERR}}$, and $\overline{\text{HALT}}$ Assertion Results	7-47
8-1	Exception Vector Assignments	8-3
8-2	Privileged Instructions	8-7
8-3	Tracing Control	8-8
8-4	Interrupt Levels and Mask Levels	8-11
8-5	Exception Priority Groups	8-16
8-6	Exception Stack Frames	8-23

LIST OF TABLES (Continued)

Table Number	Title	Page Number
9-1	cpTRAPcc Op-Mode Encodings	9-15
9-2	Coprocessor Format Word Encodings	9-17
9-3	Null Coprocessor Response Primitive Encodings	9-31
9-4	Valid Effective Address Codes	9-34
9-5	Main Processor Control Register Selector Codes	9-39
9-6	Exceptions Related to Primitive Processing	9-50
10-1	Example Instruction Stream Execution Comparison	10-7
10-2	Instruction Timings for Timing Tables	10-10
10-3	Observed Instruction Timings	10-10
11-1	Access Status Register Codes	11-4

SECTION 1
INTRODUCTION

The MC68020 is the first full 32-bit implementation of the M68000 Family of microprocessors from Motorola. Using VLSI technology, the MC68020 is implemented with 32-bit registers and data paths, 32-bit addresses, a rich instruction set, and versatile addressing modes.

The MC68020 is object code compatible with the earlier members of the M68000 Family and has the added features of new addressing modes in support of high level-languages, an on-chip instruction cache, and a flexible coprocessor interface with full IEEE floating-point support (the MC68881). The internal operations of this microprocessor operate in parallel, allowing multiple instructions to be executed concurrently.

The asynchronous bus structure of the MC68020 uses a nonmultiplexed bus with 32 bits of address and 32 bits of data. The processor supports a dynamic bus sizing mechanism that allows the processor to transfer operands to or from external devices while automatically determining device port size on a cycle-by-cycle basis. The dynamic bus interface allows access to devices of differing data bus widths, in addition to eliminating all data alignment restrictions.

1.1 FEATURES

The main features of the MC68020 are:

- Object Code Compatible with Earlier M68000 Microprocessors
- Addressing Mode Extensions for Enhanced Support of High-Level Languages
- New Bit Field Data Type Accelerates Bit-Oriented Applications, i.e., Video Graphics
- Fast On-Chip Instruction Cache Speeds Instructions and Improves Bus Bandwidth
- Coprocessor Interface to Companion 32-Bit Peripherals — the MC68881 and MC68882 Floating-Point Coprocessors and the MC68851 Paged Memory Management Unit
- Pipelined Architecture with High Degree of Internal Parallalism Allowing Multiple Instructions to be Executed Concurrently
- High-Performance Asynchronous Bus is Nonmultiplexed and Full 32 Bits
- Dynamic Bus Sizing Efficiently Supports 8-/16-/32-Bit Memories and Peripherals
- Full Support of Virtual Memory and Virtual Machine
- Sixteen 32-Bit General-Purpose Data and Address Registers
- Two 32-Bit Supervisor Stack Pointers and Five Special-Purpose Control Registers
- Eight Addressing Modes and Seven Data Types

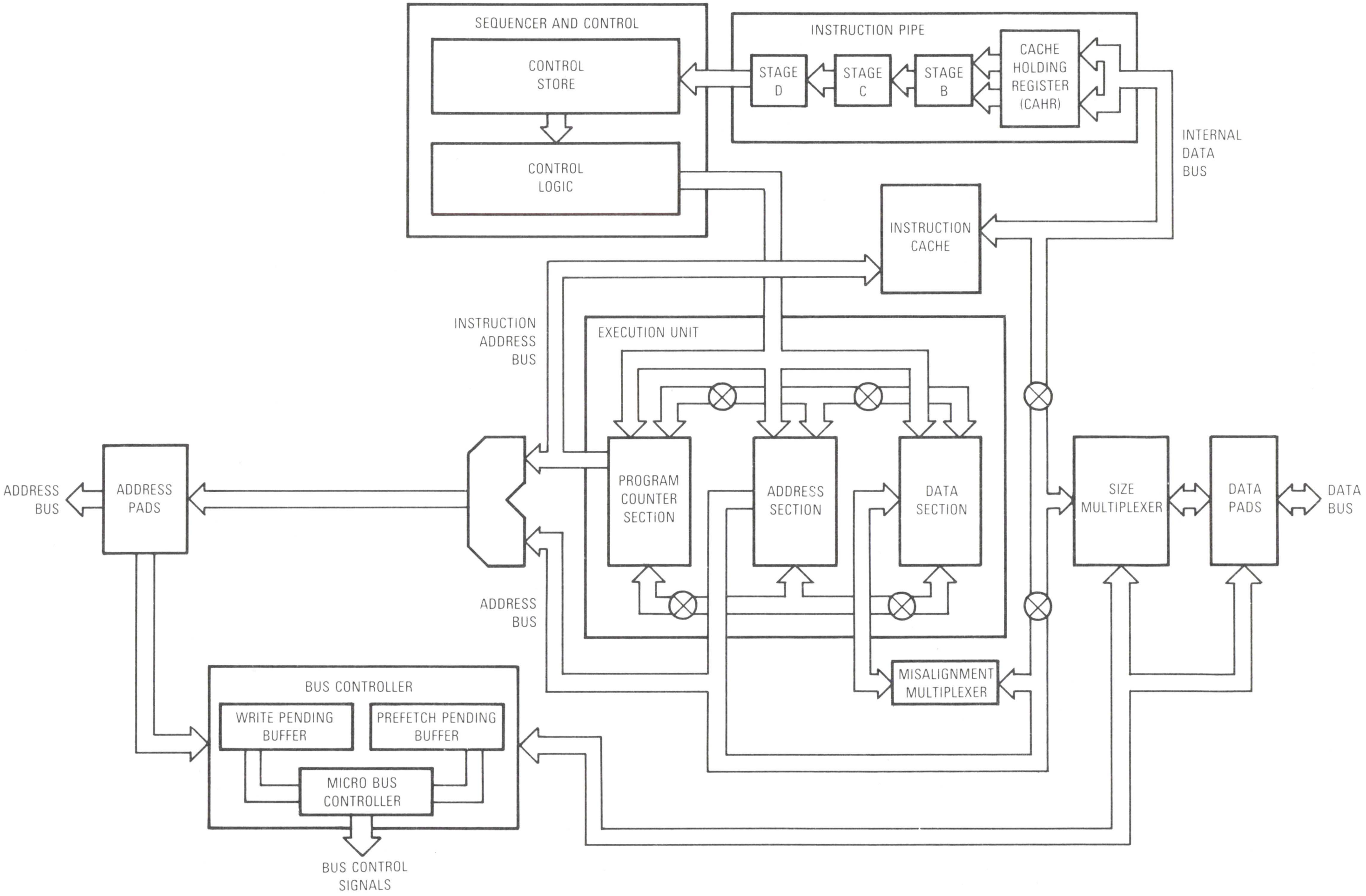

SEQUENCER AND CONTROL
CONTROL STORE
CONTROL LOGIC
INSTRUCTION PIPE
STAGE D
STAGE C
STAGE B
CACHE HOLDING REGISTER (CAHR)
INTERNAL DATA BUS
INSTRUCTION CACHE
INSTRUCTION ADDRESS BUS
EXECUTION UNIT
PROGRAM COUNTER SECTION
ADDRESS SECTION
DATA SECTION
ADDRESS BUS
ADDRESS PADS
ADDRESS BUS
SIZE MULTIPLEXER
DATA PADS
DATA BUS
MISALIGNMENT MULTIPLEXER
BUS CONTROLLER
WRITE PENDING BUFFER
PREFETCH PENDING BUFFER
MICRO BUS CONTROLLER
BUS CONTROL SIGNALS

Figure 1-1. MC68020 Block Diagram

- 4-Gigabyte Direct Addressing Range
- Selection of Processor Speeds: 12.5, 16.67, 20, 25, and 33.33 MHz

A block diagram of the MC68020 is shown in Figure 1-1.

1.2 PROGRAMMING MODEL

The programming model of the MC68020 consists of two groups of registers: the user model and the supervisor model. This corresponds to the user and supervisor privilege levels. User programs, executing at the user privilege level, can only use the registers of the user model. System software executing at the supervisor level uses the control registers of the supervisor level to perform supervisor functions.

As shown in the programming models (Figures 1-2 and 1-3) the MC68020 has 16 32-bit general-purpose registers, a 32-bit program counter, two 32-bit supervisor stack pointers, a 16-bit status register, a 32-bit vector base register, two 3-bit alternate function code registers, and two 32-bit cache handling (address and control) registers.

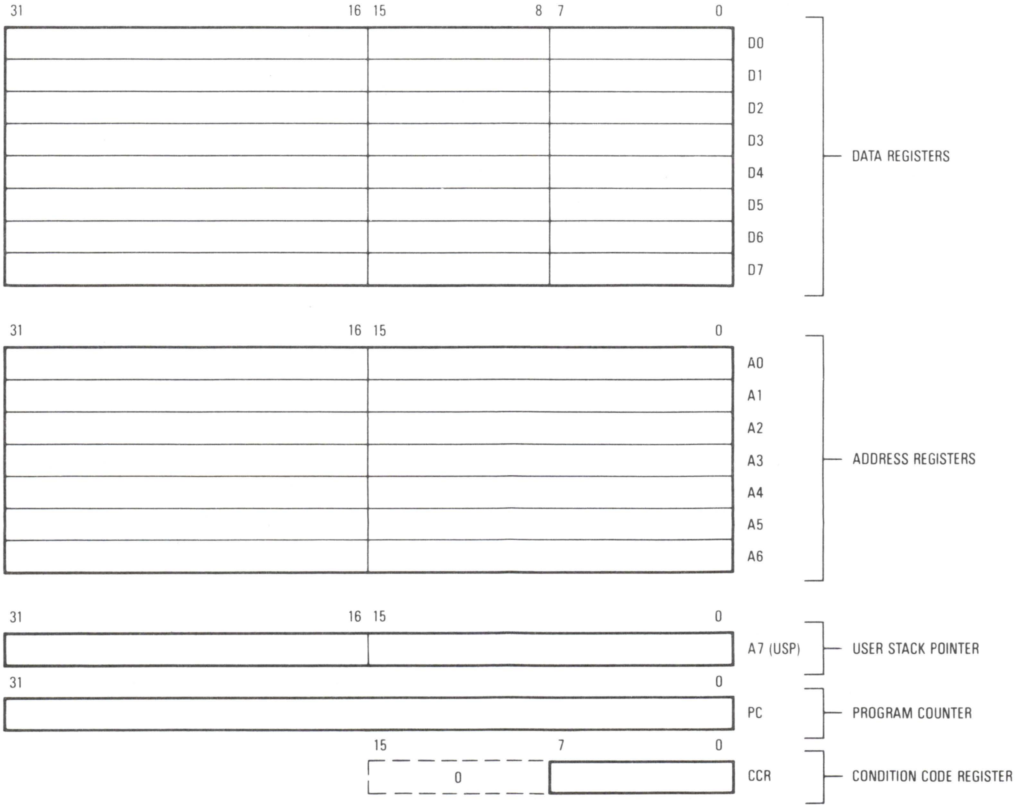

31
16 15
8 7
0
D0
D1
D2
D3
D4
D5
D6
D7
DATA REGISTERS
31
16 15
0
A0
A1
A2
A3
A4
A5
A6
ADDRESS REGISTERS
31
16 15
0
A7 (USP)
USER STACK POINTER
31
0
PC
PROGRAM COUNTER
15
7
0
0
CCR
CONDITION CODE REGISTER

Figure 1-2. User Programming Model

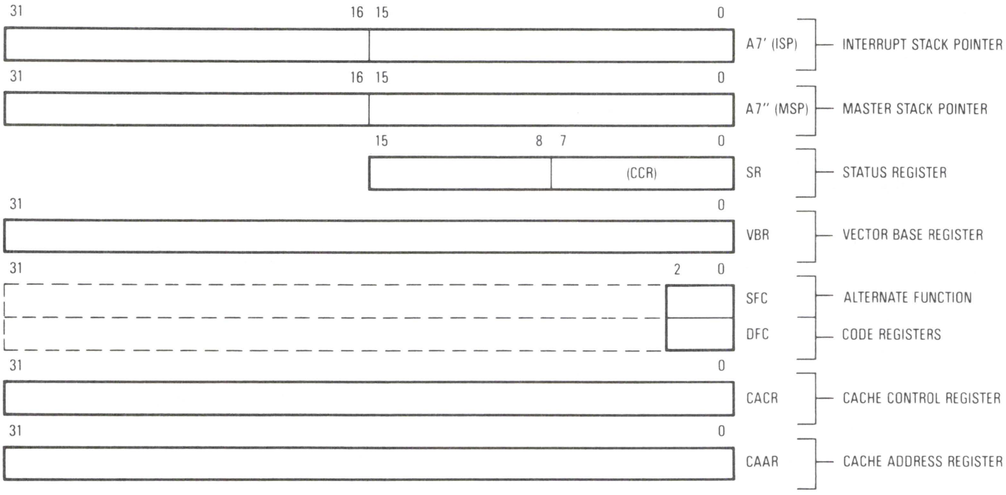

31
16 15
0
A7′ (ISP)
INTERRUPT STACK POINTER
31
16 15
0
A7″ (MSP)
MASTER STACK POINTER
15
8 7
0
(CCR)
SR
STATUS REGISTER
31
0
VBR
VECTOR BASE REGISTER
31
2 0
SFC
DFC
ALTERNATE FUNCTION
CODE REGISTERS
31
0
CACR
CACHE CONTROL REGISTER
31
0
CAAR
CACHE ADDRESS REGISTER

Figure 1-3. Supervisor Programming Model Supplement

The user programming model remains unchanged from previous M68000 Family microprocessors. The supervisor programming model supplements the user programming model and is used exclusively by the MC68020 system programmers who utilize the supervisor privilege level to implement sensitive operating system functions. Here, in the supervisor programming model, are all the controls to access and enable the special features of the MC68020. All application software, written to run at the nonprivileged user level, migrates to the MC68020 from any M68000 platform without modification.

Registers D7–D0 are used as data registers for bit and bit field (1 to 32 bits), byte (8 bit), word (16 bit), long word (32 bit), and quad word (64 bit) operations. Registers A6–A0 and the user, interrupt, and master stack pointers are address registers that may be used as software stack pointers or base address registers. Register A7 (shown as A7′, and A7″ in Figure 1-3) is a register designation that applies to the user stack pointer in the user privilege level, and to either the interrupt or master stack pointer in the supervisor privilege level. In the supervisor privilege level, the active stack pointer (interrupt or master) is called the supervisor stack pointer (SSP). In addition, the address registers may be used for word and long word operations. All of the 16 general-purpose registers (D7–D0, A7–A0) may be used as index registers.

The program counter (PC) contains the address of the next instruction to be executed by the MC68020. During instruction execution and exception processing, the processor automatically increments the contents of the PC or places a new value in the PC, as appropriate.

The status register, SR, (Figure 1-4) stores the processor status. It contains the condition codes that reflect the results of a previous operation and can be used for conditional instruction execution in a program. The condition codes are: extend (X), negative (N), zero (Z), overflow (V), and carry (C). The user byte containing the condition codes is the only

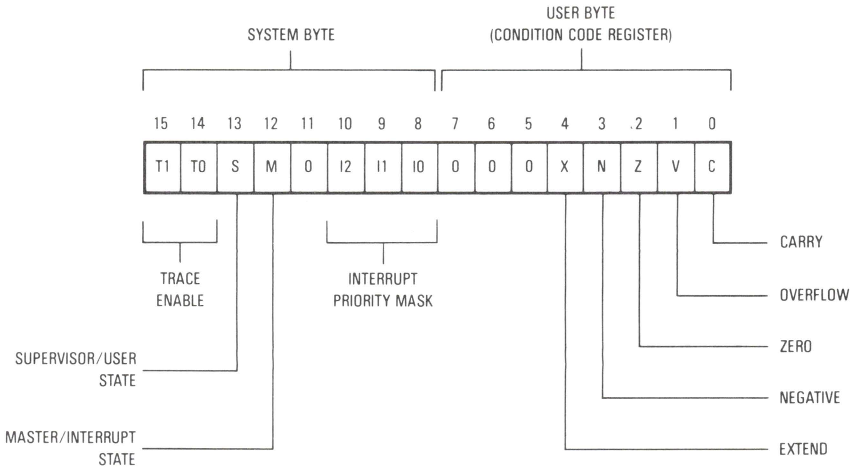

USER BYTE
(CONDITION CODE REGISTER)
SYSTEM BYTE
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
T1
T0
S
M
0
I2
I1
I0
0
0
0
X
N
Z
V
C
TRACE ENABLE
INTERRUPT PRIORITY MASK
SUPERVISOR/USER STATE
MASTER/INTERRUPT STATE
CARRY
OVERFLOW
ZERO
NEGATIVE
EXTEND

Figure 1-4. Status Register

portion of the status register information available in the user privilege level, and it is referenced as the CCR in user programs. In the supervisor privilege level, software can access the full status register, including the interrupt priority mask (three bits) as well as additional control bits. These bits indicate whether the processor is in:

1. One of Two Trace Modes (T1, T0),
2. Supervisor or User Privilege Level (S), or
3. Master or Interrupt Mode (M).

The vector base register (VBR) contains the base address of the exception vector table in memory. The displacement of an exception vector is added to the value in this register to access the vector table.

Alternate function code registers SFC and DFC contain 3-bit function codes. Function codes can be considered extensions of the 32-bit linear address that optionally provide as many as eight 4-gigabyte address spaces. Function codes are automatically generated by the processor to select address spaces for data and program at the user and supervisor privilege levels, and a CPU address space used for processor functions (for example, coprocessor communications). Registers SFC and DFC are used by certain instructions to explicitly specify the function codes for operations.

The cache control register (CACR) controls the on-chip instruction cache of the MC68020. The cache address register (CAAR) stores an address for cache control functions.

1.3 DATA TYPES AND ADDRESSING MODES

Seven basic data types are supported:

Bits
Bit Fields (Fields of consecutive bits, 1-32 bits long)

BCD Digits (Packed: 2 digits/byte, Unpacked: 1 digit/byte)
Byte Integers (8 bits)
Word Integers (16 bits)
Long Word Integers (32 bits)
Quad Word Integers (64 bits)

In addition, the instruction set supports operations on other data types such as memory addresses. The coprocessor mechanism allows direct support of floating-point operations with the MC68881 and MC68882 floating-point coprocessors, as well as specialized user-defined data types and functions.

The 18 addressing modes, shown in Table 1-1, include nine basic types:

Register Direct
Register Indirect
Register Indirect with Index
Memory Indirect
Program Counter Indirect with Displacement
Program Counter Indirect with Index
Program Counter Memory Indirect
Absolute
Immediate

The register indirect addressing modes can also postincrement, predecrement, offset, and index addresses. The program counter relative mode also has index and offset capabilities. Both modes are extended to provide indirect reference through memory. In addition to these addressing modes, many instructions implicitly specify the use of the condition code register, stack pointer, and/or program counter.

1.4 INSTRUCTION SET OVERVIEW

The instructions in the MC68020 instruction set are listed in Table 1-2. The instruction set has been tailored to support structured high-level languages and sophisticated operating systems. Many instructions operate on bytes, words, or long words and most instructions can use any of the 18 addressing modes.

1.5 VIRTUAL MEMORY AND VIRTUAL MACHINE CONCEPTS

The full addressing range of the MC68020 is 4 gigabytes (4,294,967,296 bytes) in each of eight address spaces. Even though most systems implement a smaller physical memory, the system can be made to appear to have a full 4 gigabytes of memory available to each user program by using virtual memory techniques.

In a virtual memory system, a user program can be written as if it has a large amount of memory available, when the physical memory actually present is much smaller. Similarly, a system can be designed to allow user programs to access devices that are not physically present in the system such as tape drives, disk drives, printers, terminals, and so forth. With proper software emulation, a physical system can appear to be any other M68000

Table 1-1. MC68020 Addressing Modes

Addressing Modes	Syntax
Register Direct	
Data Register Direct	Dn
Address Register Direct	An
Register Indirect	
Address Register Indirect	(An)
Address Register Indirect with Postincrement	(An)+
Address Register Indirect with Predecrement	−(An)
Address Register Indirect with Displacement	(d_{16},An)
Register Indirect with Index	
Address Register Indirect with Index (8-Bit Displacement)	(d_8,An,Xn)
Address Register Indirect with Index (Base Displacement)	(bd,An,Xn)
Memory Indirect	
Memory Indirect Postindexed	([bd,An],Xn,od)
Memory Indirect Preindexed	([bd,An,Xn],od)
Program Counter Indirect with Displacement	(d_{16},PC)
Program Counter Indirect with Index	
PC Indirect with Index (8-Bit Displacement)	(d_8,PC,Xn)
PC Indirect with Index (Base Displacement)	(bd,PC,Xn)
Program Counter Memory Indirect	
PC Memory Indirect Postindexed	([bd,PC],Xn,od)
PC Memory Indirect Preindexed	([bd,PC,Xn],od)
Absolute	
Absolute Short	(xxx).W
Absolute Long	(xxx).L
Immediate	#⟨data⟩

NOTES:

Dn = Data Register, D7–D0

An = Address Register, A7–A0

d_8, d_{16} = A twos-complement, or sign-extended displacement; added as part of the effective address calculation; size is 8 (d_8) or 16 (d_{16}) bits; when omitted, assemblers use a value of zero.

Xn = Address or data register used as an index register; form is Xn.SIZE*SCALE, where SIZE is .W or .L (indicates index register size) and SCALE is 1, 2, 4, or 8 (index register is multiplied by SCALE); use of SIZE and/or SCALE is optional.

bd = A twos-complement base displacement; when present, size can be 16 or 32 bits.

od = Outer displacement, added as part of effective address calculation after any memory indirection; use is optional with a size of 16 or 32 bits.

PC = Program Counter

⟨data⟩ = Immediate value of 8, 16, or 32 bits

( ) = Effective Address

[ ] = Use as indirect access to long word address.

computer system to a user program and the program can be given full access to all of the resources of that emulated system. Such an emulated system is called a virtual machine.

1.5.1 Virtual Memory

A system that supports virtual memory has a limited amount of high-speed physical memory that can be accessed directly by the processor and maintains an image of a much

Table 1-2. Instruction Set

Mnemonic	Description
ABCD	Add Decimal with Extend
ADD	Add
ADDA	Add Address
ADDI	Add Immediate
ADDQ	Add Quick
ADDX	Add with Extend
AND	Logical AND
ANDI	Logical AND Immediate
ASL, ASR	Arithmetic Shift Left and Right
Bcc	Branch Conditionally
BCHG	Test Bit and Change
BCLR	Test Bit and Clear
BFCHG	Test Bit Field and Change
BFCLR	Test Bit Field and Clear
BFEXTS	Signed Bit Field Extract
BFEXTU	Unsigned Bit Field Extract
BFFFO	Bit Field Find First One
BFINS	Bit Field Insert
BFSET	Test Bit Field and Set
BFTST	Test Bit Field
BKPT	Breakpoint
BRA	Branch
BSET	Test Bit and Set
BSR	Branch to Subroutine
BTST	Test Bit
CALLM	Call Module
CAS	Compare and Swap Operands
CAS2	Compare and Swap Dual Operands
CHK	Check Register Against Bound
CHK2	Check Register Against Upper and Lower Bounds
CLR	Clear
CMP	Compare
CMPA	Compare Address
CMPI	Compare Immediate
CMPM	Compare Memory to Memory
CMP2	Compare Register Against Upper and Lower Bounds
DBcc	Test Condition, Decrement and Branch
DIVS, DIVSL	Signed Divide
DIVU, DIVUL	Unsigned Divide
EOR	Logical Exclusive OR
EORI	Logical Exclusive OR Immediate
EXG	Exchange Registers
EXT, EXTB	Sign Extend
ILLEGAL	Take Illegal Instruction Trap
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link and Allocate
LSL, LSR	Logical Shift Left and Right
MOVE	Move
MOVEA	Move Address
MOVE CCR	Move Condition Code Register
MOVE SR	Move Status Register
MOVE USP	Move User Stack Pointer
MOVEC	Move Control Register
MOVEM	Move Multiple Registers
MOVEP	Move Peripheral
MOVEQ	Move Quick
MOVES	Move Alternate Address Space
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Decimal with Extend
NEG	Negate
NEGX	Negate with Extend
NOP	No Operation
NOT	Logical Complement
OR	Logical Inclusive OR
ORI	Logical Inclusive OR Immediate
ORI CCR	Logical Inclusive OR Immediate to Condition Codes
ORI SR	Logical Inclusive OR Immediate to Status Register
PACK	Pack BCD
PEA	Push Effective Address
RESET	Reset External Devices
ROL, ROR	Rotate Left and Right
ROXL, ROXR	Rotate with Extend Left and Right
RTD	Return and Deallocate
RTE	Return from Exception
RTM	Return from Module
RTR	Return and Restore Codes
RTS	Return from Subroutine
SBCD	Subtract Decimal with Extend
Scc	Set Conditionally
STOP	Stop
SUB	Subtract
SUBA	Subtract Address
SUBI	Subtract Immediate
SUBQ	Subtract Quick
SUBX	Subtract with Extend
SWAP	Swap Register Words
TAS	Test Operand and Set
TRAP	Trap
TRAPcc	Trap Conditionally
TRAPV	Trap on Overflow
TST	Test Operand
UNLK	Unlink
UNPK	Unpack BCD

COPROCESSOR INSTRUCTIONS

Mnemonic	Description
cpBcc	Branch Conditionally
cpDBcc	Test Coprocessor Condition, Decrement and Branch
cpGEN	Coprocessor General Instruction

Mnemonic	Description
cpRESTORE	Restore Internal State of Coprocessor
cpSAVE	Save Internal State of Coprocessor
cpScc	Set Conditionally
cpTRAPcc	Trap Conditionally

SECTION 2
DATA ORGANIZATION AND ADDRESSING CAPABILITIES

Most external references to memory by a microprocessor are either program references or data references; they either access instruction words or operands (data items) for an instruction. Program references are references to the program space, the section of memory that contains the program instructions and any immediate data operands that reside in the instruction stream. Refer to **SECTION 3 INSTRUCTION SET** for descriptions of the instructions in the program space. Data references refer to the data space, the section of memory that contains the program data. Data items in the instruction stream can be accessed with the program counter relative addressing modes, and these accesses are classified as program references. A third type of external reference used for coprocessor communications, interrupt acknowledge cycles, and breakpoint acknowledge cycles is classified as a CPU space reference. The MC68020 automatically sets the function codes to access the program space, the data space, or the CPU space for special functions as required.

This section describes the data organization and addressing capabilities of the MC68020. It lists the types of operands used by instructions, and describes the registers and their use as operands. Next the section describes the organization of data in memory and the addressing modes available to access data in memory. Last, the section describes the system stack and user program stacks and queues.

2.1 INSTRUCTION OPERANDS

The MC68020 supports a general purpose set of operands to serve the requirements of a large range of applications. Operands of MC68020 instructions may reside in registers, in memory, or within the instructions themselves. An instruction operand might also reside in a coprocessor. An operand may be a single bit, a bit field of from 1 to 32 bits in length, a byte (8 bits), a word (16 bits), a long word (32 bits), or a quad word (64 bits). The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation. Coprocessors are designed to support special computation models that require very specific, but widely varying data operand types and sizes. Hence, coprocessor instructions can specify operands of any size.

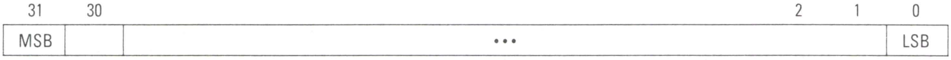
2.2 ORGANIZATION OF DATA IN REGISTERS

The eight data registers can store data operands of 1, 8, 16, 32, and 64 bits, addresses of 16 or 32 bits, or bit fields of 1 to 32 bits. The seven address registers and the three stack pointers are used for address operands of 16 or 32 bits. The control registers (SR, VBR, SFC, DFC, CACR, CAAR) vary in size according to function. Coprocessors may define unique operand sizes and support them with on-chip registers accordingly.

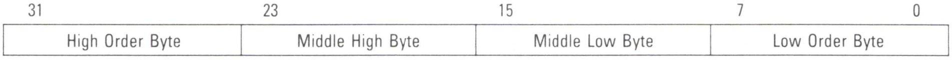
2.2.1 Data Registers

Each data register is 32 bits wide. Byte operands occupy the low order 8 bits, word operands the low order 16 bits, and long word operands the entire 32 bits. When a data register is used as either a source or destination operand, only the appropriate low-order byte or word (in byte or word operations, respectively) is used or changed; the remaining high-order portion is neither used nor changed. The least significant bit of a long word integer is addressed as bit zero and the most significant bit is addressed as bit 31. For bit fields, the most significant bit is addressed as bit zero and the least significant bit is addressed as the width of the field minus one. If the width of the field plus the offset is greater than 32, the bit field wraps around within the register. The following shows the organization of various types of data in the data registers.

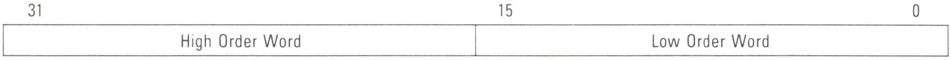
Bit (0≤Modulo (Offset)<31, Offset of 0 = MSB)


31
30
2
1
0
MSB
• • •
LSB

Byte


31
23
15
7
0
High Order Byte
Middle High Byte
Middle Low Byte
Low Order Byte

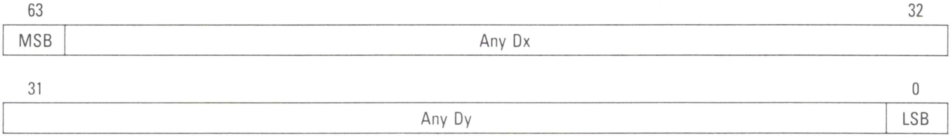
16-Bit Word


31
15
0
High Order Word
Low Order Word

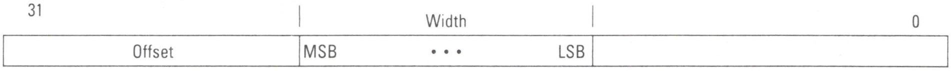
Long Word

31
0
Long Word

Quad Word

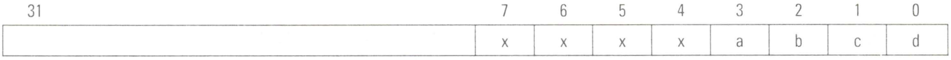

63
32
MSB
Any Dx
31
0
Any Dy
LSB

Bit Field (0≤Offset<32, 0<Width≤32)

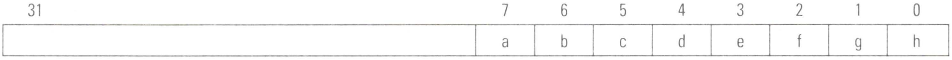

31
Width
0
Offset
MSB
• • •
LSB

Note: If width + offset<32, bit field wraps around within the register.

Unpacked BCD (a = MSB)



Packed BCD (a = MSB First Digit, e = MSB Second Digit)



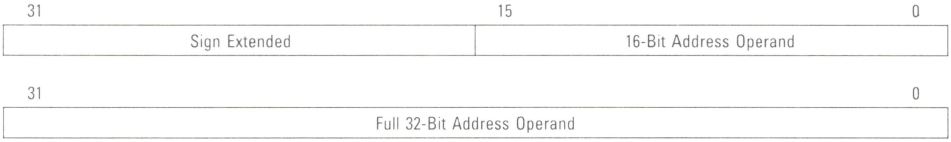
Data Organization in Data Registers

Quad word data consists of two long words: for example, the product of 32-bit multiply or the quotient of 32-bit divide operations (signed and unsigned). Quad words may be organized in any two data registers without restrictions on order or pairing. There are no explicit instructions for the management of this data type, although the MOVEM instruction can be used to move a quad word into or out of the registers.

Binary coded decimal (BCD) data represents decimal numbers in binary form. Although many BCD codes have been devised, the BCD instructions of the M68000 Family support formats in which the four least significant bits consist of a binary number having the numeric value of the corresponding decimal number. Two BCD formats are used. In the unpacked BCD format, a byte contains one digit; the four least significant bits contain the binary value and the four most significant bits are undefined. Each byte of the packed BCD format contains two digits; the least significant four bits contain the least significant digit.

2.2.2 Address Registers

Each address register and stack pointer is 32 bits wide and holds a 32-bit address. Address registers cannot be used for byte-sized operands. Therefore, when an address register is used as a source operand, either the low-order word or the entire long word operand is used, depending upon the operation size. When an address register is used as the destination operand, the entire register is affected regardless of the operation size. If the source operand is a word size, it is first sign-extended to 32 bits, and then used in the operation to an address register destination. Address registers are used primarily for addresses and to support address computation. The instruction set includes instructions that add to, subtract from, compare, and move the contents of address registers. The following shows the organization of addresses in address registers.



Address Organization in Address Registers

2.2.3 Control Registers

The control registers described in this section contain control information for supervisor functions and vary in size. With the exception of the user portion of the status register (condition code register) they are accessed only by instructions at the supervisor privilege level.

The status register (SR), shown in Figure 1-4, is 16 bits wide. Only 12 bits of the status register are defined; all undefined values are reserved by Motorola for future definition. The undefined bits are read as zeros and should be written as zeros for future compatibility. The lower byte of the status register is the condition code register (CCR). Operations to the CCR can be performed at the supervisor or user privilege level. All operations to the status register and condition code register are word-size operations, but for all CCR operations the upper byte is read as all zeros and is ignored when written, regardless of privilege level.

The supervisor programming model (Figure 1-3) shows the control registers. The cache control register (CACR) provides control and status information for the on-chip instruction cache. The cache address register (CAAR) contains the address for cache control functions. The vector base register (VBR) provides the base address of the exception vector table. All operations involving the CACR, CAAR, and VBR are long word operations, whether these registers are used as the source or the destination operand.

The alternate function code registers (SFC and DFC) are 32-bit registers with only bits 2:0 implemented that contain the address space values (FC2–FC0) for the read or write operand of the MOVES instruction. The MOVEC instruction is used to transfer values to and from the alternate function code registers. These are long word transfers; the upper 29 bits are read as zeros and are ignored when written.

2.3 ORGANIZATION OF DATA IN MEMORY

Memory is organized on a byte-addressable basis where lower addresses correspond to higher-order bytes. The address, N, of a long word data item corresponds to the address of the most significant byte of the highest-order word. The lower-order word is located at address N + 2, leaving the least significant byte at address N + 3 (refer to Figure 2-1). Notice that the MC68020 does not require ***data*** to be aligned on word boundaries (refer to Figure 2-2) but the most efficient data transfers occur when data is aligned on the same byte boundary as its operand size. However, ***instruction*** words must be aligned on word boundaries.

The data types supported in memory by the MC68020 are: bit and bit field data; integer data of 8, 16, or 32 bits; 32-bit addresses; and binary coded decimal data (packed and unpacked). These data types are organized in memory as shown in Figure 2-2. Note that all of these data types can be accessed at any byte address.

Coprocessors can implement any data types and lengths up to 255 bytes. For example, the MC68881/MC68882 floating-point coprocessors support memory accesses for quad-word sized items (double-precision floating-point values).

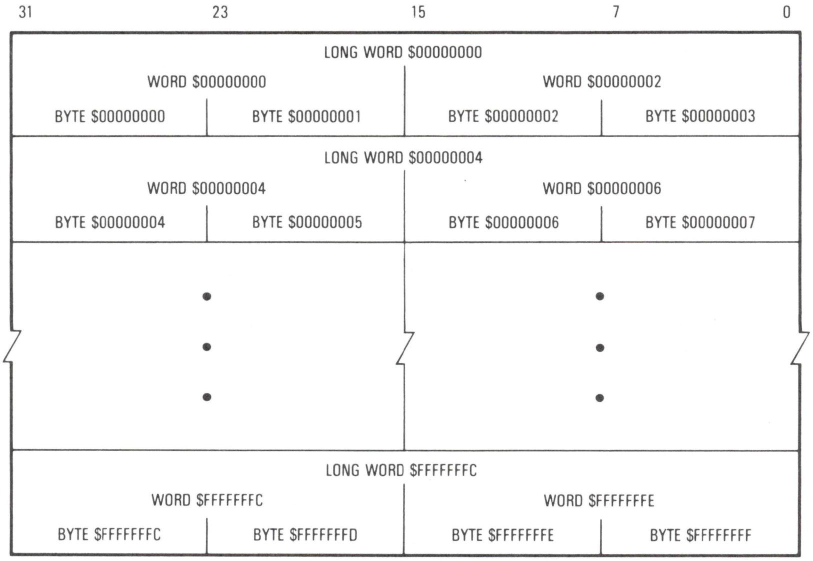

31
23
15
7
0
LONG WORD $00000000
WORD $00000000
WORD $00000002
BYTE $00000000
BYTE $00000001
BYTE $00000002
BYTE $00000003
LONG WORD $00000004
WORD $00000004
WORD $00000006
BYTE $00000004
BYTE $00000005
BYTE $00000006
BYTE $00000007
LONG WORD $FFFFFFFC
WORD $FFFFFFFC
WORD $FFFFFFFE
BYTE $FFFFFFFC
BYTE $FFFFFFFD
BYTE $FFFFFFFE
BYTE $FFFFFFFF

Figure 2-1. Memory Operand Address

A bit operand is specified by a base address that selects one byte in memory (the base byte) and a bit number that selects one bit in this byte. The most significant bit of the byte is bit seven.

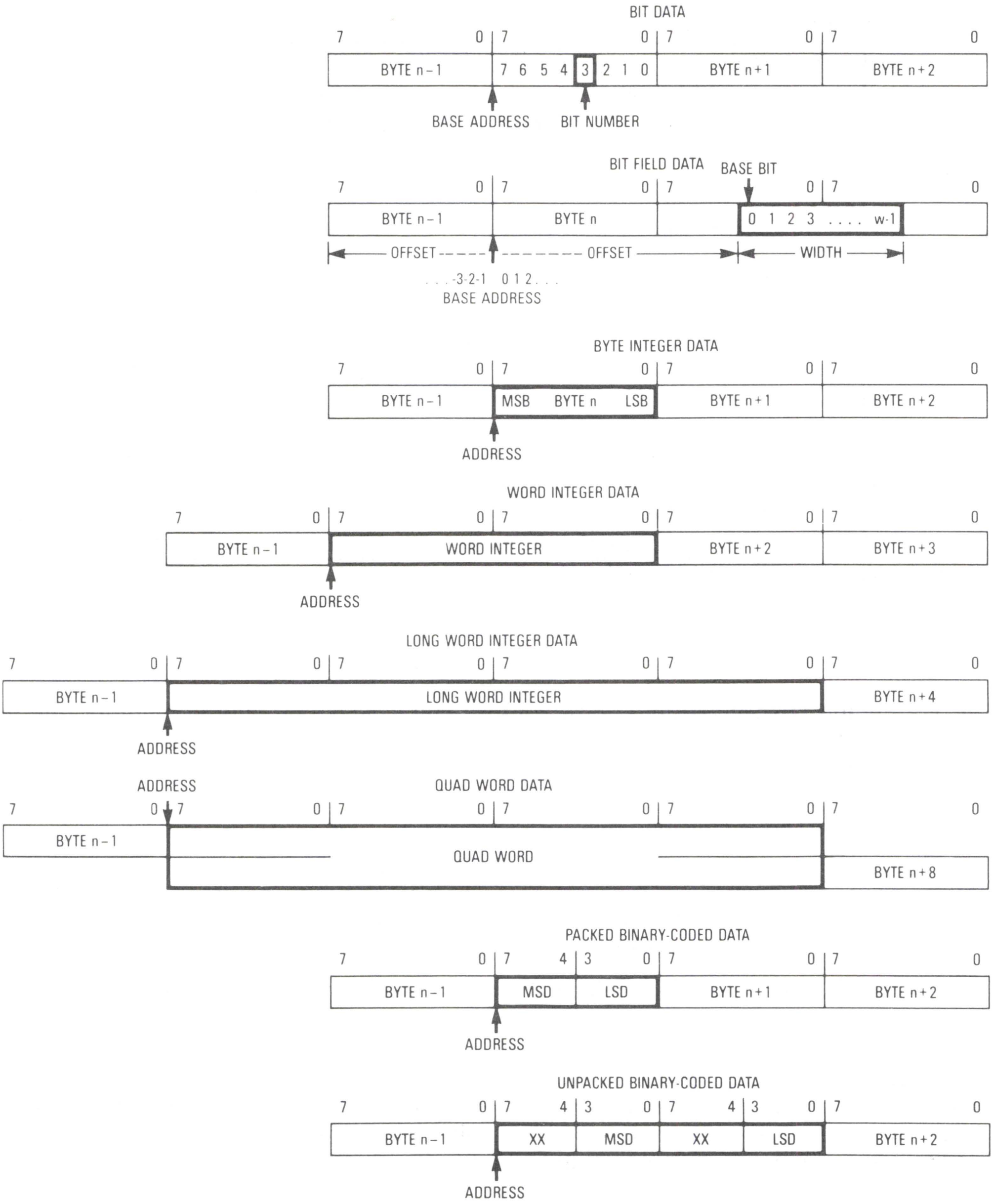
A bit field operand is specified by:

1. A base address that selects one byte in memory,
2. A bit field offset that indicates the leftmost (base) bit of the bit field in relation to the most significant bit of the base byte, and
3. A bit field width that determines how many bits to the right of the base bit are in the bit field.

The most significant bit of the base byte is bit field offset 0, the least significant bit of the base byte is bit field offset 7, and the least significant bit of the previous byte in memory is bit offset -1. Bit field offsets may have values in the range of -2^{31} to $2^{31}-1$, and bit field widths may range between 1 and 32 bits.

2.4 ADDRESSING MODES

The addressing mode of an instruction can specify the value of an operand (with an immediate operand), or a register that contains the operand (with the register direct addressing mode), or how the effective address of an operand in memory is derived. An assembler syntax has been defined for each addressing mode.


BIT DATA
7 0 7 0 7 0 7 0
BYTE n−1
7 6 5 4 3 2 1 0
BYTE n+1
BYTE n+2
BASE ADDRESS
BIT NUMBER
BIT FIELD DATA
BASE BIT
BYTE n−1
BYTE n
0 1 2 3 .... w-1
OFFSET
OFFSET
WIDTH
...-3-2-1 0 1 2...
BASE ADDRESS
BYTE INTEGER DATA
MSB BYTE n LSB
ADDRESS
WORD INTEGER DATA
WORD INTEGER
BYTE n+2
BYTE n+3
ADDRESS
LONG WORD INTEGER DATA
LONG WORD INTEGER
BYTE n+4
ADDRESS
ADDRESS
QUAD WORD DATA
QUAD WORD
BYTE n+8
PACKED BINARY-CODED DATA
7 4 3 0
MSD
LSD
ADDRESS
UNPACKED BINARY-CODED DATA
XX
MSD
XX
LSD
ADDRESS

XX = USER-DEFINED VALUE

Figure 2-2. Memory Data Organization

Figure 2-3 shows the general format of the single-effective-address instruction operation word. The effective address field specifies the addressing mode for an operand that can use one of the numerous defined modes. The ⟨ea⟩ designation is composed of two 3-bit fields: the mode field and the register field. The value in the mode field selects one, or a set of addressing modes. The register field specifies a register for the mode, or a submode for modes that do not use registers.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	X	X	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Figure 2-3. Single-Effective-Address Instruction Operation Word

Many instructions imply the addressing mode for one of the operands. The formats of these instructions include appropriate fields for operands that use only one addressing mode.

The effective address field may require additional information to fully specify the operand address. This additional information, called the effective address extension, is contained in an additional word or words and is considered part of the instruction. Refer to **2.5 EFFECTIVE ADDRESS ENCODING SUMMARY** for a description of the extension word formats.

The notational conventions used in the addressing mode descriptions in this section are:

EA—Effective address

An—Address register n
    Example: A3 is address register 3

Dn—Data register n
    Example: D5 is data register 5

Xn.SIZE*SCALE—Denotes index register n (data or address), the index size (W for word, L for long word), and a scale factor (1, 2, 4, or 8, for no, word, long word, or 8 for quad word scaling, respectively).

PC—The program counter

d_n—Displacement value, n bits wide

bd—Base displacement

od—Outer displacement

L—Long word size

W—Word size

( )—Identify an indirect address in a register

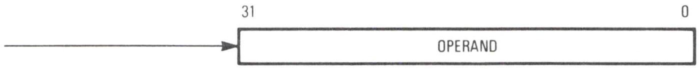
[ ]—Identify an indirect address in memory

When the addressing mode uses a register, the register field of the operation word specifies the register to be used. Other fields within the instruction specify whether the register selected is an address or data register and how the register is to be used.

2.4.1 Data Register Direct Mode

In the data register direct mode, the operand is in the data register specified by the effective address register field.

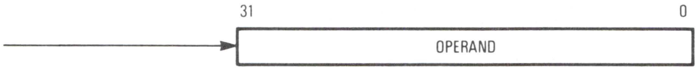
GENERATION: EA = Dn
ASSEMBLER SYNTAX: Dn
MODE: 000
REGISTER: n
DATA REGISTER: Dn
NUMBER OF EXTENSION WORDS: 0



2.4.2 Address Register Direct Mode

In the address register direct mode, the operand is in the address register specified by the effective address register field.

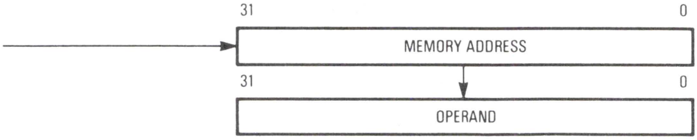
GENERATION: EA = An
ASSEMBLER SYNTAX: An
MODE: 001
REGISTER: n
ADDRESS REGISTER: An
NUMBER OF EXTENSION WORDS: 0



2.4.3 Address Register Indirect Mode

In the address register indirect mode, the operand is in memory and the address of the operand is in the address register specified by the register field.

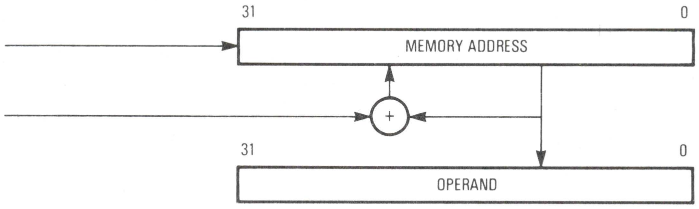
GENERATION: EA = (An)
ASSEMBLER SYNTAX: (An)
MODE: 010
REGISTER: n
ADDRESS REGISTER: An
MEMORY ADDRESS:
NUMBER OF EXTENSION WORDS: 0



2.4.4 Address Register Indirect with Postincrement Mode

In the address register indirect with postincrement mode, the operand is in memory, and the address of the operand is in the address register specified by the register field. After the operand address is used, it is incremented by one, two, or four depending on the size of the operand: byte, word, or long word. Coprocessors may support incrementing for any size of operand, up to 255 bytes. If the address register is the stack pointer and the operand size is byte, the address is incremented by two rather than one to keep the stack pointer aligned to a word boundary.

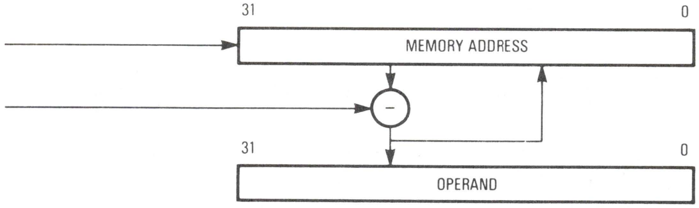
GENERATION: EA = (An)
An = An + SIZE
ASSEMBLER SYNTAX: (An) +
MODE: 011
REGISTER: n
ADDRESS REGISTER: An
OPERAND LENGTH (1, 2, OR 4):
MEMORY ADDRESS:
NUMBER OF EXTENSION WORDS: 0



2.4.5 Address Register Indirect with Predecrement Mode

In the address register indirect with predecrement mode, the operand is in memory and the address of the operand is in the address register specified by the register field. Before the operand address is used, it is decremented by one, two, or four depending on the operand size: byte, word, or long word. Coprocessors may support decrementing for any operand size up to 255 bytes. If the address register is the stack pointer and the operand size is byte, the address is decremented by two rather than one to keep the stack pointer aligned to a word boundary.

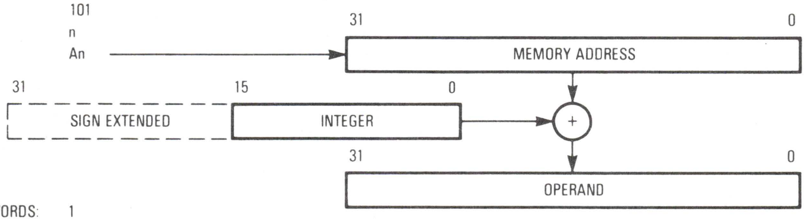
GENERATION: An = An − SIZE
EA = (An)
ASSEMBLER SYNTAX: − (An)
MODE: 100
REGISTER: n
ADDRESS REGISTER: An
OPERAND LENGTH (1, 2, OR 4):
MEMORY ADDRESS:
NUMBER OF EXTENSION WORDS: 0



2.4.6 Address Register Indirect with Displacement Mode

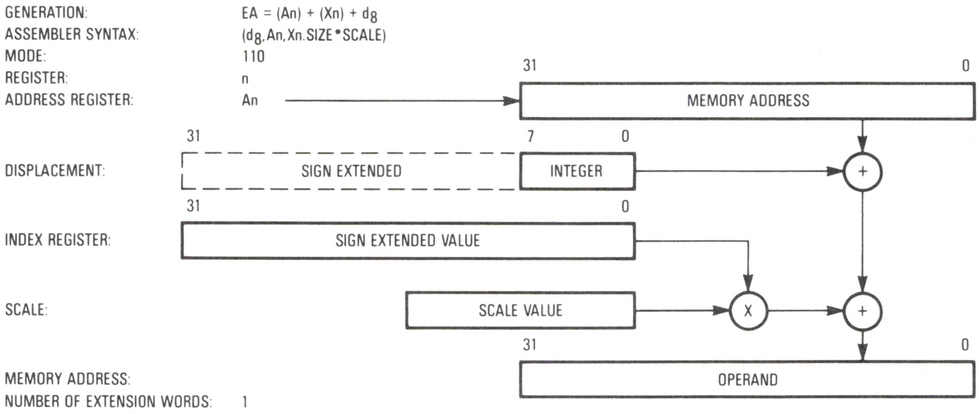
In the address register indirect with displacement mode, the operand is in memory. The address of the operand is the sum of the address in the address register plus the sign-extended 16-bit displacement integer in the extension word. Displacements are always sign extended to 32 bits prior to being used in effective address calculations.

GENERATION: EA = (An) + d_{16}
ASSEMBLER SYNTAX: (d_{16},An)
MODE: 101
REGISTER: n
ADDRESS REGISTER: An
DISPLACEMENT:
MEMORY ADDRESS:
NUMBER OF EXTENSION WORDS: 1



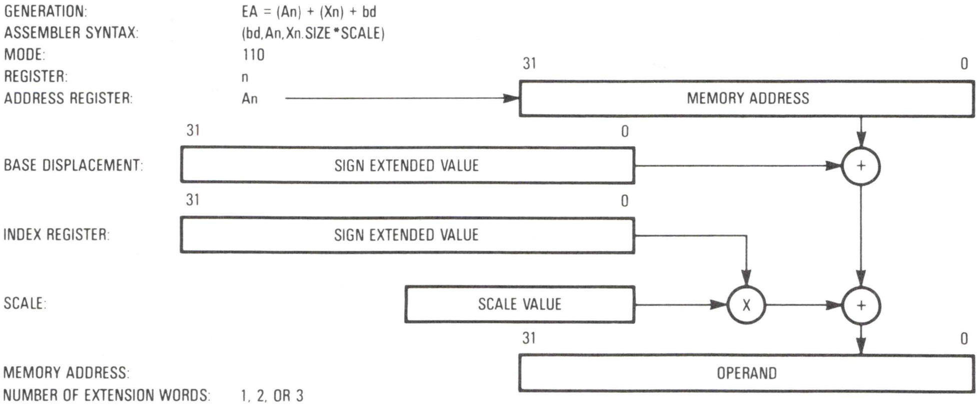
2.4.7 Address Register Indirect with Index (8-Bit Displacement) Mode

This addressing mode requires one extension word that contains the index register indicator and an 8-bit displacement. The index register indicator includes size and scale information. In this mode, the operand is in memory. The address of the operand is the sum of the contents of the address register, the sign extended displacement value in the low-order eight bits of the extension word, and the sign extended contents of the index register (possibly scaled). The user must specify the displacement, the address register, and the index register in this mode.


GENERATION:
EA = (An) + (Xn) + d8
ASSEMBLER SYNTAX:
(d8,An,Xn.SIZE*SCALE)
MODE:
110
REGISTER:
n
ADDRESS REGISTER:
An
31
0
MEMORY ADDRESS
31
7
0
DISPLACEMENT:
SIGN EXTENDED
INTEGER
+
31
0
INDEX REGISTER:
SIGN EXTENDED VALUE
SCALE:
SCALE VALUE
X
+
31
0
MEMORY ADDRESS:
OPERAND
NUMBER OF EXTENSION WORDS:
1

2.4.8 Address Register Indirect with Index (Base Displacement) Mode

This addressing mode requires an index register indicator and an optional 16- or 32-bit sign-extended base displacement. The index register indicator includes size and scaling information. The operand is in memory. The address of the operand is the sum of the contents of the address register, the scaled contents of the sign-extended index register, and the base displacement.

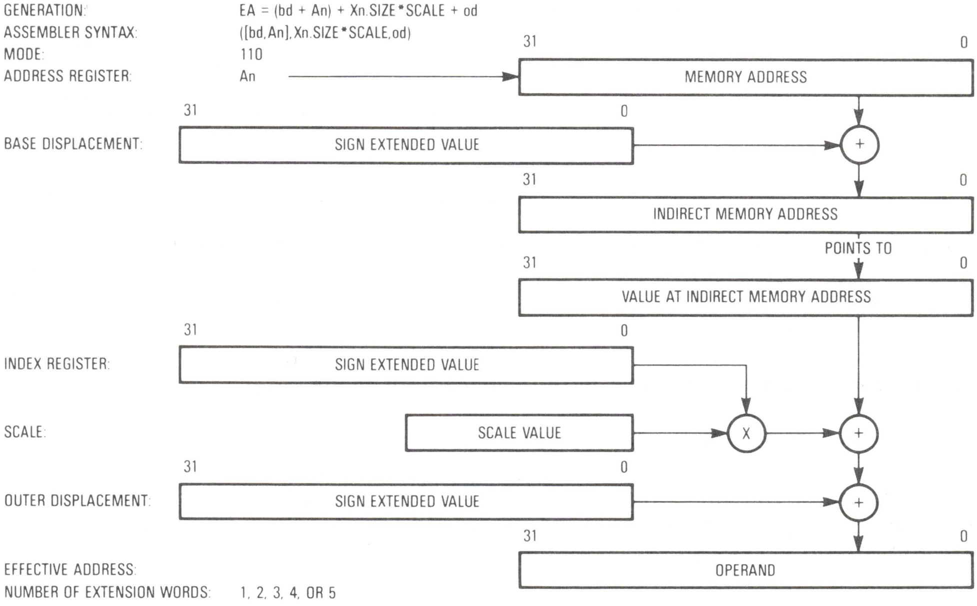

GENERATION:
EA = (An) + (Xn) + bd
ASSEMBLER SYNTAX:
(bd,An,Xn.SIZE*SCALE)
MODE:
110
REGISTER:
n
ADDRESS REGISTER:
An
31
0
MEMORY ADDRESS
31
0
BASE DISPLACEMENT:
SIGN EXTENDED VALUE
+
31
0
INDEX REGISTER:
SIGN EXTENDED VALUE
SCALE:
SCALE VALUE
X
+
31
0
MEMORY ADDRESS:
OPERAND
NUMBER OF EXTENSION WORDS:
1, 2, OR 3

In this mode, the address register, the index register, and the displacement are all optional. If none is specified, the effective address is zero. This mode provides a data register indirect address when no address register is specified and the index register is a data register (Dn).

2.4.9 Memory Indirect Postindexed Mode

In this mode, the operand and its address are in memory. The processor calculates an intermediate indirect memory address using the base register (An) and base displacement (bd). The processor accesses a long word at this address and adds the index operand (Xn.SIZE*SCALE) and the outer displacement to yield the effective address. Both displacements and the index register contents are sign extended to 32 bits.

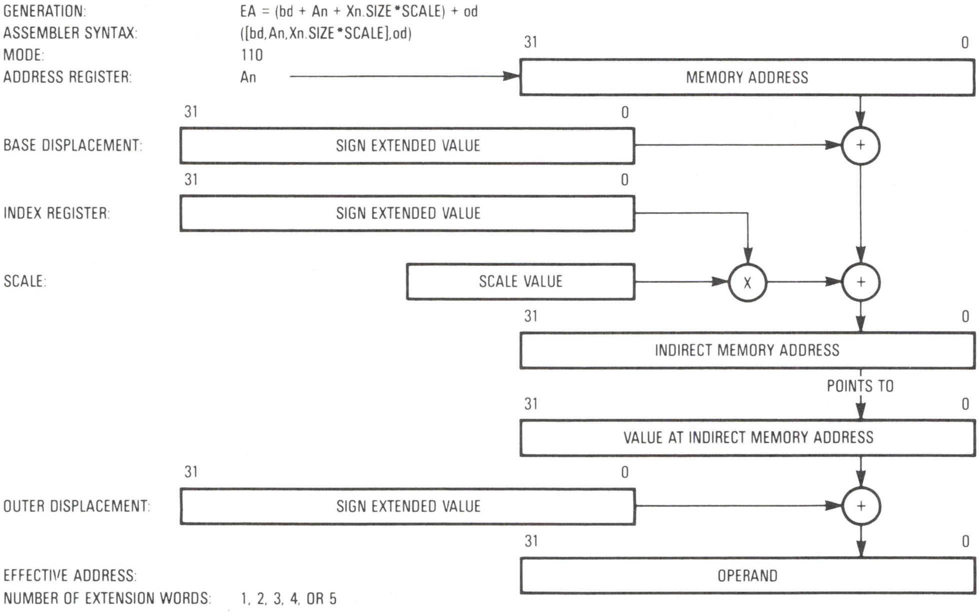
In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.


GENERATION:
EA = (bd + An) + Xn.SIZE*SCALE + od
ASSEMBLER SYNTAX:
([bd,An],Xn.SIZE*SCALE,od)
MODE:
110
ADDRESS REGISTER:
An
31
0
MEMORY ADDRESS
BASE DISPLACEMENT:
31
0
SIGN EXTENDED VALUE
+
31
0
INDIRECT MEMORY ADDRESS
POINTS TO
31
0
VALUE AT INDIRECT MEMORY ADDRESS
INDEX REGISTER:
31
0
SIGN EXTENDED VALUE
SCALE:
SCALE VALUE
X
+
OUTER DISPLACEMENT:
31
0
SIGN EXTENDED VALUE
+
EFFECTIVE ADDRESS:
31
0
OPERAND
NUMBER OF EXTENSION WORDS:
1, 2, 3, 4, OR 5

2.4.10 Memory Indirect Preindexed Mode

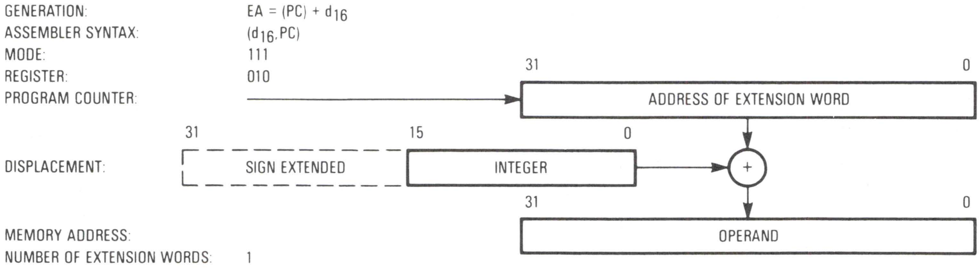
In this mode, the operand and its address are in memory. The processor calculates an intermediate indirect memory address using the base register (An), a base displacement (bd), and the index operand (Xn.SIZE*SCALE). The processor accesses a long word at this address and adds the outer displacement to yield the effective address. Both displacements and the index register contents are sign extended to 32 bits.

In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.


GENERATION: EA = (bd + An + Xn.SIZE*SCALE) + od
ASSEMBLER SYNTAX: ([bd,An,Xn.SIZE*SCALE],od)
MODE: 110
ADDRESS REGISTER: An
31
0
MEMORY ADDRESS
BASE DISPLACEMENT:
SIGN EXTENDED VALUE
INDEX REGISTER:
SIGN EXTENDED VALUE
SCALE:
SCALE VALUE
X
+
INDIRECT MEMORY ADDRESS
POINTS TO
VALUE AT INDIRECT MEMORY ADDRESS
OUTER DISPLACEMENT:
SIGN EXTENDED VALUE
EFFECTIVE ADDRESS:
OPERAND
NUMBER OF EXTENSION WORDS: 1, 2, 3, 4, OR 5

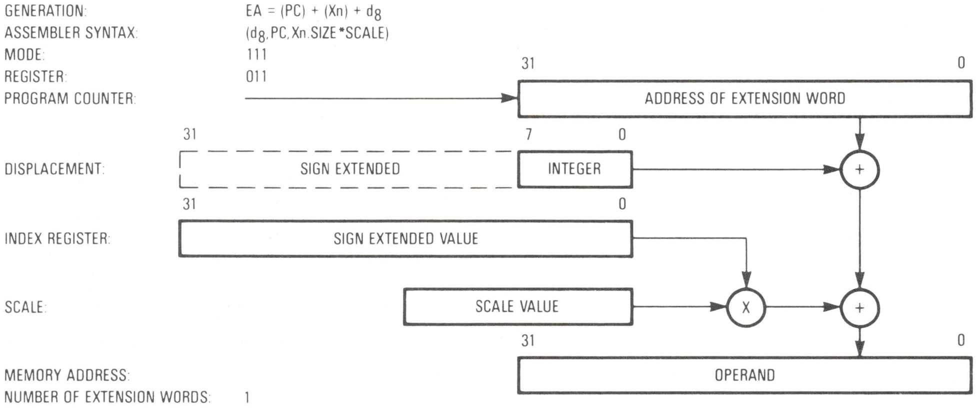
2.4.11 Program Counter Indirect with Displacement Mode

In this mode, the operand is in memory. The address of the operand is the sum of the address in the program counter and the sign-extended 16-bit displacement integer in the extension word. The value in the program counter is the address of the extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 ADDRESS SPACE TYPES**).


GENERATION: EA = (PC) + d16
ASSEMBLER SYNTAX: (d16,PC)
MODE: 111
REGISTER: 010
PROGRAM COUNTER:
31
0
ADDRESS OF EXTENSION WORD
DISPLACEMENT:
31
15
0
SIGN EXTENDED
INTEGER
+
MEMORY ADDRESS:
OPERAND
NUMBER OF EXTENSION WORDS: 1

2.4.12 Program Counter Indirect with Index (8-Bit Displacement)

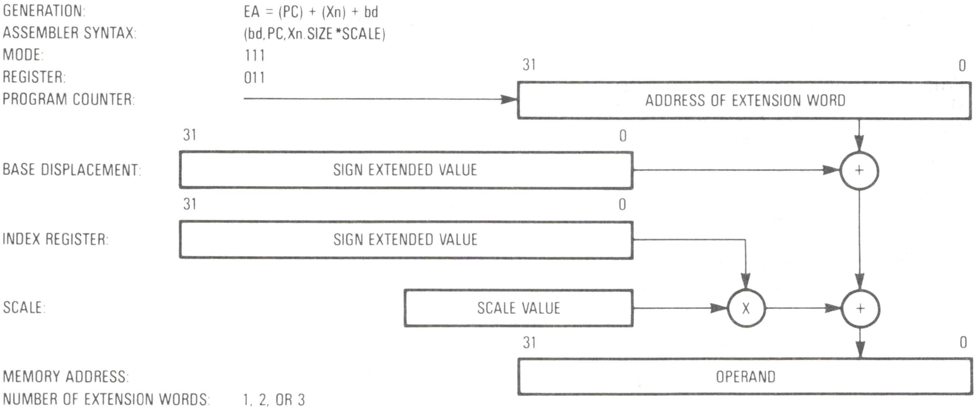
This mode is similar to the address register indirect with index (8-bit displacement) mode described in **2.4.7 Address Register Indirect with Index (8-Bit Displacement Mode)**, but the PC is used as the base register. The operand is in memory. The address of the operand is the sum of the address in the program counter, the sign-extended displacement integer in the lower eight bits of the extension word, and the sized, scaled, and sign-extended index operand. The value in the PC is the address of the extension word. This reference is a program space reference and is only allowed for reads. The user must include the displacement, the PC, and the index register when specifying this addressing mode.


GENERATION:
EA = (PC) + (Xn) + d_8
ASSEMBLER SYNTAX:
(d_8,PC,Xn.SIZE*SCALE)
MODE:
111
REGISTER:
011
PROGRAM COUNTER:
31
0
ADDRESS OF EXTENSION WORD
31
7
0
DISPLACEMENT:
SIGN EXTENDED
INTEGER
+
31
0
INDEX REGISTER:
SIGN EXTENDED VALUE
SCALE:
SCALE VALUE
X
+
31
0
MEMORY ADDRESS:
OPERAND
NUMBER OF EXTENSION WORDS:
1

2.4.13 Program Counter Indirect with Index (Base Displacement) Mode

This mode is similar to the address register indirect with index (base displacement) mode described in **2.4.8 Address Register Indirect with Index (Base Displacement Mode)**, but the PC is used as the base register. It requires an index register indicator and an optional 16- or 32-bit sign-extended base displacement. The operand is in memory. The address of the operand is the sum of the contents of the PC, the scaled contents of the sign-extended index register, and the base displacement. The value of the PC is the address of the first extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 ADDRESS SPACE TYPES**).

In this mode, the program counter, the index register, and the displacement are all optional. However, the user must supply the assembler notation "ZPC" (zero value is taken for the PC) to indicate that the PC is not used. This allows the user to access the program space, without using the PC in calculating the effective address. The user can access the program space with a data register indirect access by placing ZPC in the instruction and specifying a data register (Dn) as the index register.

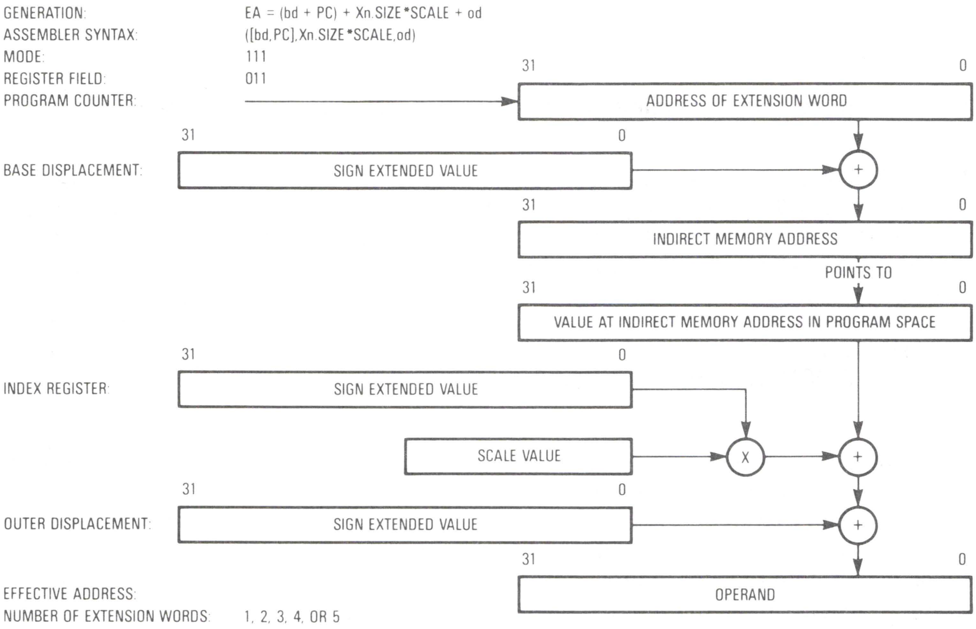


GENERATION: EA = (PC) + (Xn) + bd
ASSEMBLER SYNTAX: (bd,PC,Xn.SIZE*SCALE)
MODE: 111
REGISTER: 011
NUMBER OF EXTENSION WORDS: 1, 2, OR 3

PROGRAM COUNTER:
31
0
ADDRESS OF EXTENSION WORD
BASE DISPLACEMENT:
SIGN EXTENDED VALUE
INDEX REGISTER:
SIGN EXTENDED VALUE
SCALE:
SCALE VALUE
X
+
MEMORY ADDRESS:
OPERAND

2.4.14 Program Counter Memory Indirect Postindexed Mode

This mode is similar to the memory indirect postindexed mode described in **2.4.9 Memory Indirect Postindexed Mode**, but the PC is used as the base register. Both the operand and operand address are in memory. The processor calculates an intermediate indirect memory address by adding a base displacement (bd) to the PC contents. The processor accesses a long word at that address and adds the scaled contents of the index register and the optional outer displacement (od) to yield the effective address. The value of the PC used in the calculation is the address of the first extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 ADDRESS SPACE TYPES**).



GENERATION: EA = (bd + PC) + Xn.SIZE*SCALE + od
ASSEMBLER SYNTAX: ([bd,PC],Xn.SIZE*SCALE,od)
MODE: 111
REGISTER FIELD: 011
NUMBER OF EXTENSION WORDS: 1, 2, 3, 4, OR 5

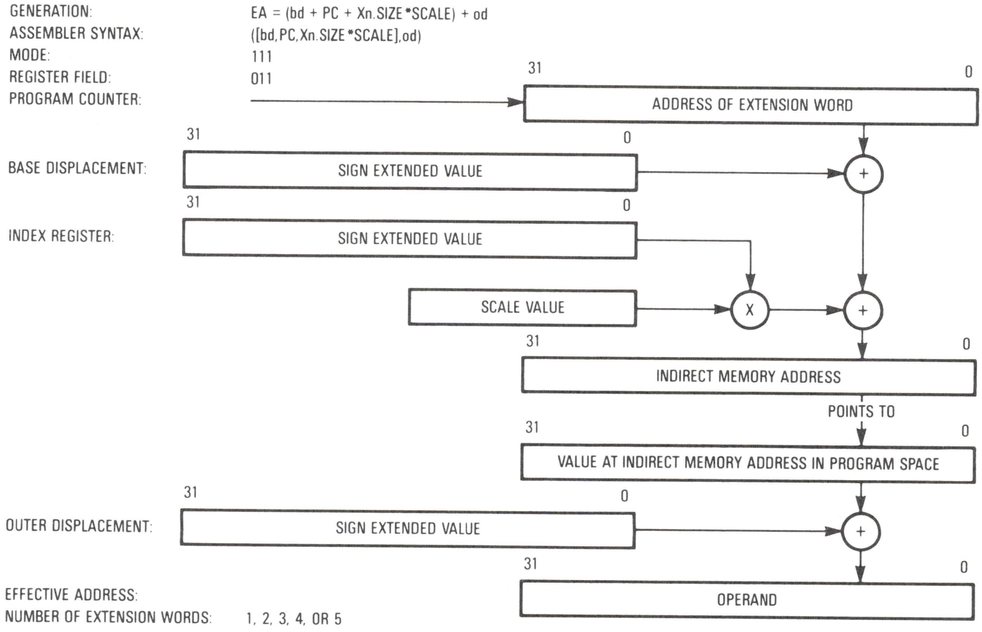
PROGRAM COUNTER:
31
0
ADDRESS OF EXTENSION WORD
BASE DISPLACEMENT:
SIGN EXTENDED VALUE
+
INDIRECT MEMORY ADDRESS
POINTS TO
VALUE AT INDIRECT MEMORY ADDRESS IN PROGRAM SPACE
INDEX REGISTER:
SIGN EXTENDED VALUE
SCALE VALUE
X
OUTER DISPLACEMENT:
SIGN EXTENDED VALUE
EFFECTIVE ADDRESS:
OPERAND

In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. However, the user must supply the assembler notation ZPC (zero value is taken for the PC) to indicate that the PC is not used. This allows the user to access the program space, without using the PC in calculating the effective address. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.

2.4.15 Program Counter Memory Indirect Preindexed Mode

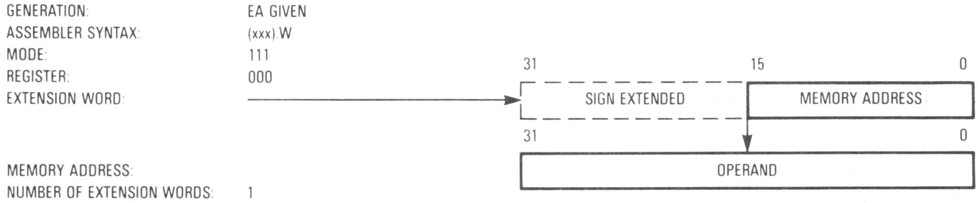
This mode is similar to the memory indirect preindexed mode described in **2.4.10 Memory Indirect Preindexed Mode**, but the PC is used as the base register. Both the operand and operand address are in memory. The processor calculates an intermediate indirect memory address by adding the PC contents, a base displacement (bd), and the scaled contents of an index register. The processor accesses a long word at that address and adds the optional outer displacement (od) to yield the effective address. The value of the PC is the address of the first extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 ADDRESS SPACE TYPES**).

In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. However, the user must supply the assembler notation ZPC (zero value is taken for the PC) to indicate that the PC is not used. This allows the user to access the program space, without using the PC in calculating the effective address. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.


GENERATION: EA = (bd + PC + Xn.SIZE*SCALE) + od
ASSEMBLER SYNTAX: ([bd,PC,Xn.SIZE*SCALE],od)
MODE: 111
REGISTER FIELD: 011
PROGRAM COUNTER:
31
0
ADDRESS OF EXTENSION WORD
BASE DISPLACEMENT:
SIGN EXTENDED VALUE
+
INDEX REGISTER:
SIGN EXTENDED VALUE
SCALE VALUE
X
+
INDIRECT MEMORY ADDRESS
POINTS TO
VALUE AT INDIRECT MEMORY ADDRESS IN PROGRAM SPACE
OUTER DISPLACEMENT:
SIGN EXTENDED VALUE
+
EFFECTIVE ADDRESS:
OPERAND
NUMBER OF EXTENSION WORDS: 1, 2, 3, 4, OR 5

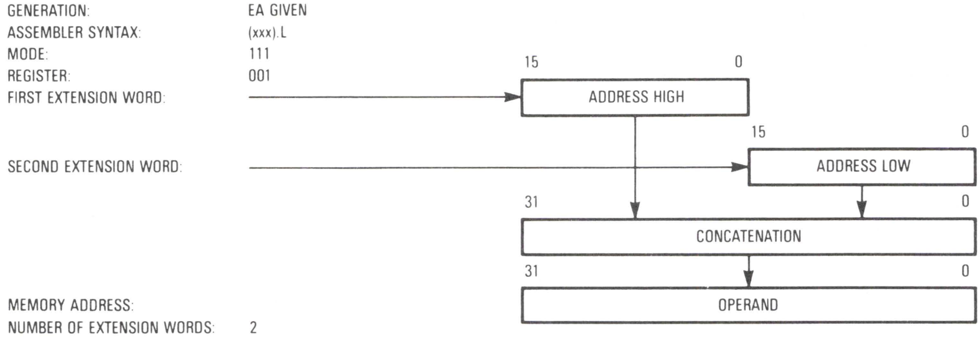
2.4.16 Absolute Short Address Mode

In this addressing mode, the operand is in memory and the address of the operand is in the extension word. The 16-bit address is sign extended to 32 bits before it is used.


GENERATION: EA GIVEN
ASSEMBLER SYNTAX: (xxx).W
MODE: 111
REGISTER: 000
EXTENSION WORD:
31
15
0
SIGN EXTENDED
MEMORY ADDRESS
31
0
OPERAND
MEMORY ADDRESS:
NUMBER OF EXTENSION WORDS: 1

2.4.17 Absolute Long Address Mode

In this mode, the operand is in memory and the address of the operand occupies the two extension words following the instruction word in memory. The first extension word contains the high order part of the address; the low order part of the address is the second extension word.


GENERATION: EA GIVEN
ASSEMBLER SYNTAX: (xxx).L
MODE: 111
REGISTER: 001
FIRST EXTENSION WORD:
15
0
ADDRESS HIGH
15
0
SECOND EXTENSION WORD:
ADDRESS LOW
31
0
CONCATENATION
31
0
OPERAND
MEMORY ADDRESS:
NUMBER OF EXTENSION WORDS: 2

2.4.18 Immediate Data

In this addressing mode, the operand is in one or two extension words:

Byte Operation
  Operand is in the low-order byte of the extension word

Word Operation
  Operand is in the extension word

Long Word Operation
  The high-order 16 bits of the operand are in the first extension word; the low-order 16 bits are in the second extension word.

Coprocessor instructions can support immediate data of any size. The instruction word is followed by as many extension words as are required.

Generation:	Operand given
Assembler Syntax:	#xxx
Mode Field:	111
Register Field:	100
Number of Extension Words:	1 or 2, except for coprocessor instructions

2.5 EFFECTIVE ADDRESS ENCODING SUMMARY

Most of the addressing modes use one of the three formats shown in Table 2-1. The single-effective-address instruction is in the format of the instruction word. The encoding of the mode field of this word selects the addressing mode. The register field contains the general register number or a value that selects the addressing mode when the mode field contains "111". Table 2-3 shows the encoding of these fields. Some indexed or indirect modes use

Table 2-1. Effective Address Specification Formats

Single-Effective-Address Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	X	X	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Brief Format Extension Word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	REGISTER			W/L	SCALE		0	DISPLACEMENT							

Full Format Extension Word(s)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	REGISTER			W/L	SCALE		1	BS	IS	BD SIZE		0	I/IS		
BASE DISPLACEMENT (0, 1, OR 2 WORDS)															
OUTER DISPLACEMENT (0, 1, OR 2 WORDS)															

Field	Definition
Instruction:	
Register	General Register Number
Extensions:	
Register	Index Register Number
D/A	Index Register Type 0 = Dn 1 = An
W/L	Word/Long Word Index Size 0 = Sign Extended Word 1 = Long Word
Scale	Scale Factor 00 = 1 01 = 2 10 = 4 11 = 8

Field	Definition
BS	Base Register Suppress: 0 = Base Register Added 1 = Base Register Suppressed
IS	Index Suppress: 0 = Evaluate and Add Index Operand 1 = Suppress Index Operand
BD SIZE	Base Displacement Size: 00 = Reserved 01 = Null Displacement 10 = Word Displacement 11 = Long Displacement
I/IS	Index/Indirect Selection: Indirect and Indexing Operand Determined in Conjunction with Bit 6, Index Suppress

the instruction word followed by the brief format extension word. Other indexed or indirect modes consist of the instruction word and the full format of extension words. The longest instruction for the MC68020 contains ten extension words. It is a MOVE instruction with full format extension words for both the source and destination effective addresses, and with 32-bit base displacements and 32-bit outer displacements for both addresses. However, coprocessor instructions can have any number of extension words. Refer to the coprocessor instruction formats in **SECTION 9 COPROCESSOR INTERFACE DESCRIPTION**.

For effective addresses that use the full format, the index suppress (IS) bit and the index/indirect selection (I/IS) field determine the type of indexing and indirection. Table 2-2 lists the indexing and indirection operations corresponding to all combinations of IS and I/IS values.

Table 2-2. IS-I/IS Memory Indirection Encodings

IS	Index/Indirect	Operation
0	000	No Memory Indirection
0	001	Indirect Preindexed with Null Outer Displacement
0	010	Indirect Preindexed with Word Outer Displacement
0	011	Indirect Preindexed with Long Outer Displacement
0	100	Reserved
0	101	Indirect Postindexed with Null Outer Displacement
0	110	Indirect Postindexed with Word Outer Displacement
0	111	Indirect Postindexed with Long Outer Displacement
1	000	No Memory Indirection
1	001	Memory Indirect with Null Outer Displacement
1	010	Memory Indirect with Word Outer Displacement
1	011	Memory Indirect with Long Outer Displacement
1	100–111	Reserved

Effective address modes are grouped according to the use of the mode. They can be classified as follows:

Data	A data addressing effective address mode is one that refers to data operands.
Memory	A memory addressing effective address mode is one that refers to memory operands.
Alterable	An alterable addressing effective address mode is one that refers to alterable (writable) operands.
Control	A control addressing effective address mode is one that refers to memory operands without an associated size.

Table 2-3 shows the categories to which each of the effective addressing modes belong.

These categories are sometimes combined, forming new categories that are more restrictive. Two combined classifications are alterable memory or data alterable. The former refers to those addressing modes that are both alterable and memory addresses, and the latter refers to addressing modes that are both data and alterable.

Table 2-3. Effective Addressing Mode Categories

Address Modes	Mode	Register	Data	Memory	Control	Alterable	Assembler Syntax
Data Register Direct	000	reg. no.	X	—	—	X	Dn
Address Register Direct	001	reg. no.	—	—	—	X	An
Address Register Indirect	010	reg. no.	X	X	X	X	(An)
Address Register Indirect with Postincrement	011	reg. no.	X	X	—	X	(An)+
Address Register Indirect with Predecrement	100	reg. no.	X	X	—	X	−(An)
Address Register Indirect with Displacement	101	reg. no.	X	X	X	X	(d_{16},An)
Address Register Indirect with Index (8-Bit Displacement)	110	reg. no.	X	X	X	X	(d_8,An,Xn)
Address Register Indirect with Index (Base Displacement)	110	reg. no.	X	X	X	X	(bd,An,Xn)
Memory Indirect Postindexed	110	reg. no.	X	X	X	X	([bd,An],Xn,od)
Memory Indirect Preindexed	110	reg. no.	X	X	X	X	([bd,An,Xn],od)
Absolute Short	111	000	X	X	X	X	(xxx).W
Absolute Long	111	001	X	X	X	X	(xxx).L
Program Counter Indirect with Displacement	111	010	X	X	X	—	(d_{16},PC)
Program Counter Indirect with Index (8-Bit) Displacement	111	011	X	X	X	—	(d_8,PC,Xn)
Program Counter Indirect with Index (Base Displacement)	111	011	X	X	X	—	(bd,PC,Xn)
PC Memory Indirect Postindexed	111	011	X	X	X	—	([bd,PC],Xn,od
PC Memory Indirect Preindexed	111	011	X	X	X	—	([bd,PC,Xn],od)
Immediate	111	100	X	X	—	—	#⟨data⟩

2.6 PROGRAMMER'S VIEW OF ADDRESSING MODES

Extensions to the indexed addressing modes, indirection, and full 32-bit displacements provide additional programming capabilities for the MC68020. The following paragraphs describes addressing techniques that exploit these capabilities and summarizes the addressing modes from a programming point of view.

Several of the addressing techniques described use data registers and address registers interchangeably. While the MC68020 provides this capability, its performance has been optimized for addressing with address registers. The performance of a program that uses address registers in address calculations is superior to that of a program that similarly uses data registers. The specification of addresses with data registers should be used sparingly (if at all), particularly in programs that require maximum performance.

2.6.1 Addressing Capabilities

In the MC68020, setting the base register suppress (BS) bit in the full format extension word (Table 2-1) suppresses use of the base address register in calculating the effective address. This allows any index register to be used in place of the base register. Since any of the data registers can be index registers, this provides a data register indirect form (Dn). The mode could be called register indirect (Rn), since either a data register or an address register can be used. This addressing mode is an extension to the M68000 Family because the MC68020 can use both the data registers and the address registers to address memory. The capability of specifying the size and scale of an index register (Xn.SIZE*SCALE) in these modes provides additional addressing flexibility. Using the SIZE parameter, either the entire contents of the index register can be used, or the least significant word can be sign extended to provide a 32-bit index value (refer to Figure 2-4).

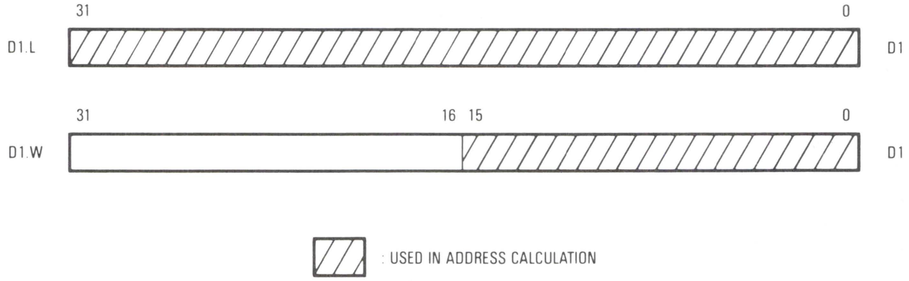


Figure 2-4. Using SIZE in the Index Selection

For the MC68020, the register indirect modes can be extended further. Since displacements can be 32 bits wide, they can represent absolute addresses or the results of expressions that contain absolute addresses. This allows the general register indirect form to be (bd,Rn) or (bd,An,Rn) when the base register is not suppressed. Thus, an absolute address can be directly indexed by one or two registers (refer to Figure 2-5).

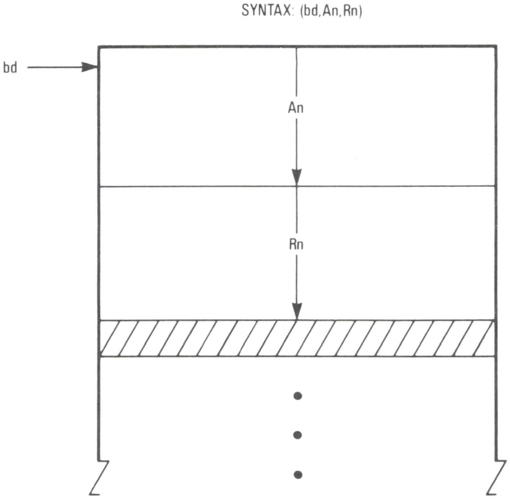
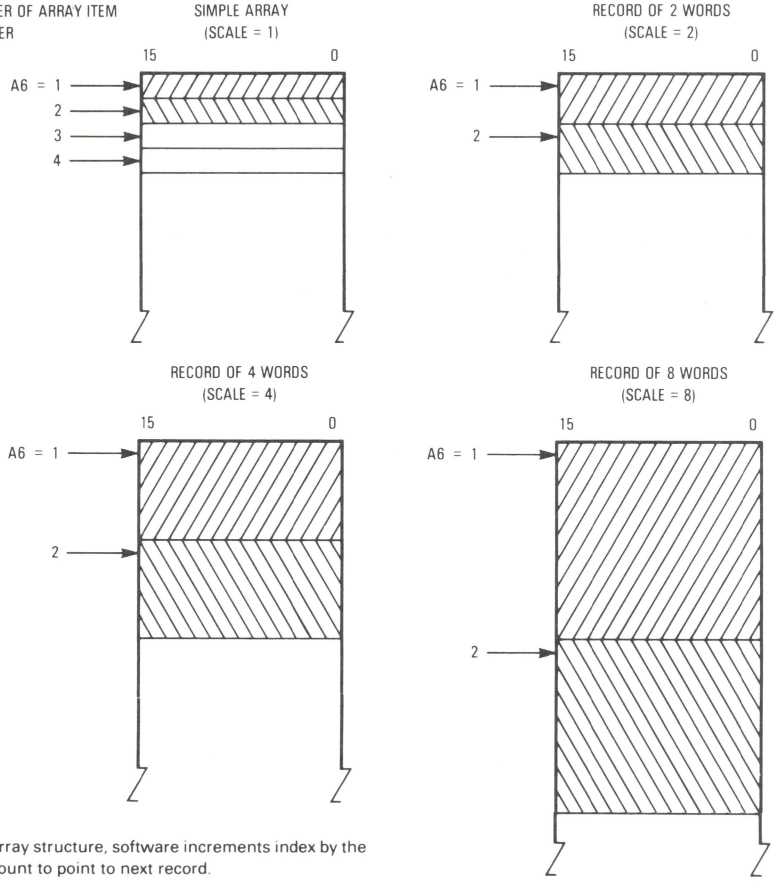


Figure 2-5. Using Absolute Address with Indexes

Scaling provides an optional shifting of the value in an index register to the left by zero, one, two, or three bits before using it in the effective address calculation (the actual value in the index register remains unchanged). This is equivalent to multiplying the register by one, two, four, or eight for direct subscripting into an array of elements of corresponding size using an arithmetic value residing in any of the 16 general registers. Scaling does not add to the effective address calculation time. However, when combined with the appropriate derived modes, it produces additional capabilities. Arrayed structures can be addressed absolutely and then subscripted, for example (bd,Rn*scale). Optionally, an address register that contains a dynamic displacement can be included in the address calculation (bd,An,Rn*scale). Another variation that can be derived is (An,Rn*scale). In the first case, the array address is the sum of the contents of a register and a displacement, as shown in Figure 2-6. In the second example, An contains the address of an array and Rn contains a subscript.

SYNTAX: MOVE.W (A5,A6.L*SCALE),(A7)
WHERE:

A5 = ADDRESS OF ARRAY STRUCTURE
A6 = INDEX NUMBER OF ARRAY ITEM
A7 = STACK POINTER



NOTE: Regardless of array structure, software increments index by the appropriate amount to point to next record.

Figure 2-6. Addressing Array Items

The memory indirect addressing modes use a long word pointer in memory to access an operand. Any of the modes previously described can be used to address the memory pointer. Because the base and index registers can both be suppressed, the displacement acts as an absolute address, providing indirect absolute memory addressing (refer to Figure 2-7).

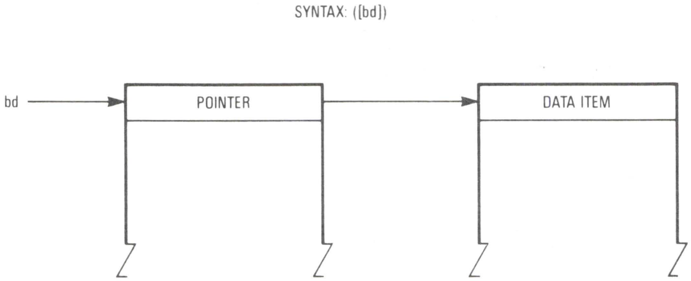


Figure 2-7. Using Indirect Absolute Memory Addressing

The outer displacement (od) available in the memory indirect modes is added to the pointer in memory. The syntax for these modes is ([bd,An],Xn,od) and ([bd,An,Xn],od). When the pointer is the address of a structure in memory and the outer displacement is the offset of an item in the structure, the memory indirect modes can access the item efficiently (refer to Figure 2-8).

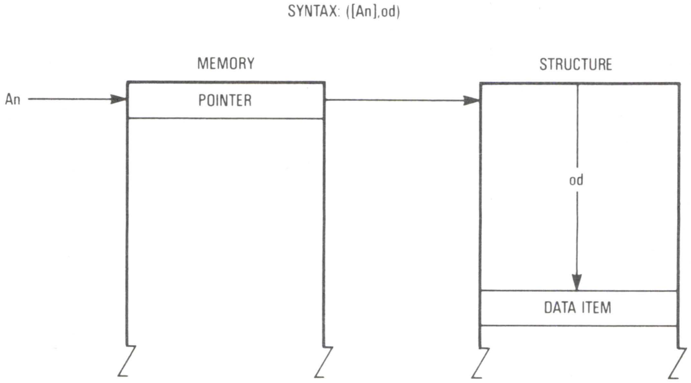


Figure 2-8. Accessing an Item in a Structure Using Pointer

Memory indirect addressing modes are used with a base displacement in five basic forms:

1. [bd,An] — Indirect, suppressed index register
2. ([bd,An,Xn]) — Preindexed indirect
3. ([bd,An],Xn) — Postindexed indirect
4. ([bd,An,Xn],od) — Preindexed indirect with outer displacement
5. ([bd,An],Xn,od — Postindexed indirect with outer displacement

The indirect, suppressed index register mode, Figure 2-9, uses the contents of register An as an index to the pointer located at the address specified by the displacement. The actual data item is at the address in the selected pointer.

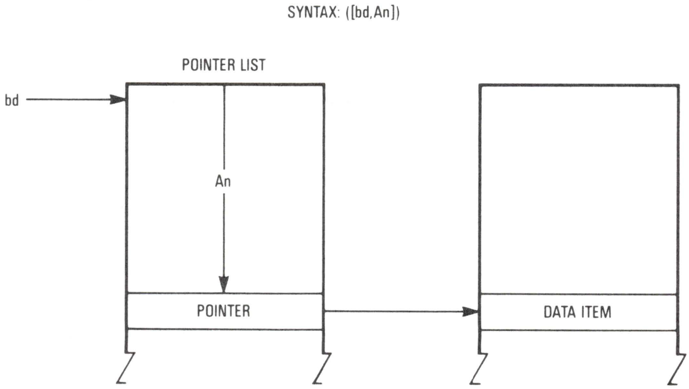


Figure 2-9. Indirect Addressing, Suppressed Index Register

The preindexed indirect mode, Figure 2-10, uses the contents of An as an index to the pointer list structure at the displacement. Register Xn is the index to the pointer, which contains the address of the data item.

The postindexed indirect mode, Figure 2-11, uses the contents of An as an index to the pointer list at the displacement. Register Xn is used as an index to the structure of data items located at the address specified by the pointer. Figure 2-12 shows the preindexed indirect addressing with outer displacement mode.

The postindexed indirect mode with outer displacement, Figure 2-13, uses the contents of An as an index to the pointer list at the displacement. Register Xn is used as an index to the structure of data structures at the address in the pointer. The outer displacement (od) is the displacement of the data item within the selected data structure.

2.6.2 General Addressing Mode Summary

The addressing modes described in the previous paragraphs are derived from specific combinations of options in the indexing mode, or a selection of two alternate addressing

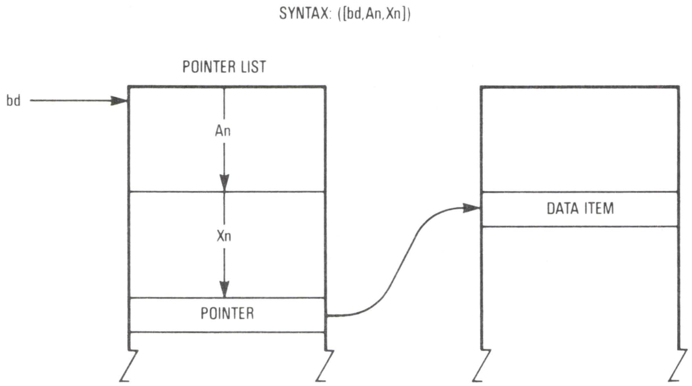

SYNTAX: ([bd,An,Xn])
POINTER LIST
bd
An
Xn
POINTER
DATA ITEM

Figure 2-10. Preindexed Indirect Addressing

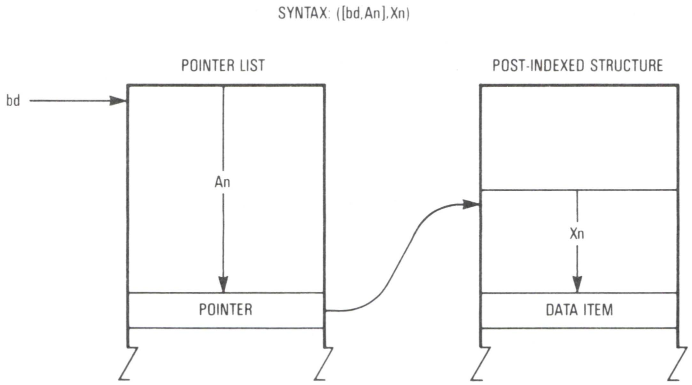

SYNTAX: ([bd,An],Xn)
POINTER LIST
POST-INDEXED STRUCTURE
bd
An
POINTER
Xn
DATA ITEM

Figure 2-11. Postindexed Indirect Addressing

modes. For example, the addressing mode called register indirect (Rn) assembles as the address register indirect if the register is an address register. If Rn is a data register, the assembler uses the address register indirect with index mode using the data register as the indirect register and suppresses the address register by setting the base suppress bit in the effective address specification. Assigning an address register as Rn provides higher performance than using a data register as Rn. Another case is (bd,An) which selects an addressing mode depending on the size of the displacement. If the displacement is 16 bits or less, the address register indirect with displacement mode (d_{16},An) is used. When a 32-bit displacement is required, the address register indirect with index (bd,An,Xn) is used with the index register suppressed.

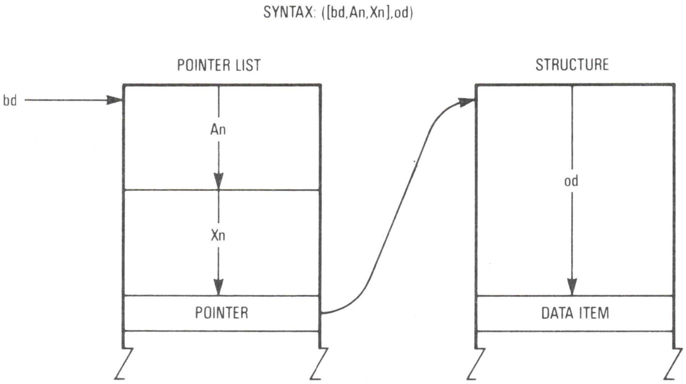

SYNTAX: ([bd,An,Xn],od)
POINTER LIST
STRUCTURE
bd
An
Xn
POINTER
od
DATA ITEM

Figure 2-12. Preindexed Indirect with Outer Displacement

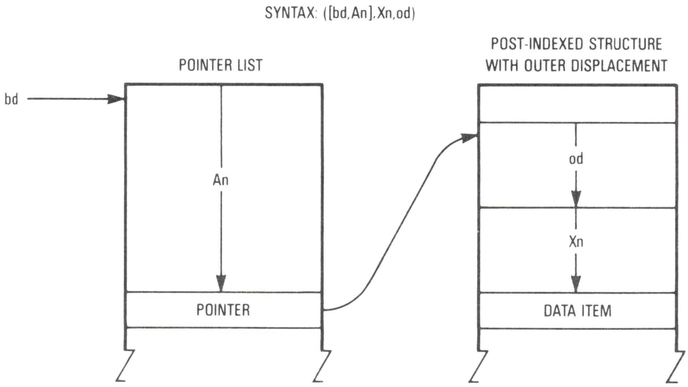

SYNTAX: ([bd,An],Xn,od)
POINTER LIST
POST-INDEXED STRUCTURE
WITH OUTER DISPLACEMENT
bd
An
POINTER
od
Xn
DATA ITEM

Figure 2-13. Postindexed Indirect Addressing with Outer Displacement

It is useful to examine the derived addressing modes available to a programmer (without regard to the MC68020 effective addressing mode actually encoded) because the programmer need not be concerned about these decisions. The assembler can choose the more efficient addressing mode to encode.

In the list of derived addressing modes that follows, common programming terms are used. These definitions apply:

pointer	— Long word value in a register or in memory which represents an address.
base	— A pointer combined with a displacement to represent an address.

index — A constant or variable value added into an effective address calculation. A constant index is a displacement. A variable index is always represented by a register containing the value.

disp — Displacement, a constant index.

subscript — The use of any of the data or address registers as a variable index subscript into arrays of items 1, 2, 4, or 8 bytes in size.

relative — An address calculated from the program counter contents. The address is position independent and is in program space. All other addresses but psaddr are in data space.

addr — An absolute address.

psaddr — An absolute address in program space. All other addresses but PC relative are in data space.

preindexed — All modes from absolute address through program counter relative.

postindexed — Any of the following modes:
addr — Absolute address in data space.
psaddr,ZPC — Absolute address in program space.
An — Register pointer.
disp,An — Register pointer with constant displace-
addr,An — ment.
disp,PC — Absolute address with single variable name.
Simple PC relative.

The addressing modes defined in programming terms which are derivations of the addressing modes provided by the MC68020 architecture are:

Immediate Data — #data:
The data is a constant located in the instruction stream.

Register Direct — Rn:
The contents of a register is the operand.

Scanning Modes:
(An)+
Address register pointer automatically incremented after use.
–(An)
Address register pointer automatically decremented before use.

Absolute Address:
(addr)
Absolute address in data space.
(psaddr,ZPC)
Absolute address in program space. Symbol ZPC suppresses the PC, but retains PC-relative mode to directly access the program space.

Register Pointer:
(Rn)
Register as a pointer.
(disp,Rn)
Register as a pointer with constant index (or base address).

Indexing:

- (An,Rn)
  - Register pointer An with variable index Rn.
- (disp,An,Rn)
  - Register pointer with constant and variable index (or a base address with a variable index).
- (addr,Rn)
  - Absolute address with variable index.
- (addr,An,Rn)
  - Absolute address with 2 variable indexes.

Subscripting:

- (An,Rn*scale)
  - Address register pointer subscript.
- (disp,An,Rn*scale)
  - Address register pointer subscript with constant displacement (or base address with subscript).
- (addr, Rn*scale)
  - Absolute address with subscript.
- (addr,An,Rn*scale)
  - Absolute address subscript with variable index.

Program Relative:

- (disp,PC)
  - Simple PC relative.
- (disp,PC,Rn)
  - PC relative with variable index.
- (disp,PC,Rn*scale)
  - PC relative with subscript.

Memory Pointer:

- ([preindexed])
  - Memory pointer directly to data operand.
- ([preindexed],disp)
  - Memory pointer as base with displacement to data operand.
- ([postindexed],Rn)
  - Memory pointer with variable index.
- ([postindexed],disp,Rn)
  - Memory pointer with constant and variable index.
- ([postindexed],Rn*scale)
  - Memory pointer subscripted.
- ([postindexed], disp, Rn*scale)
  - Memory pointer subscripted with constant index.

2.7 M68000 FAMILY ADDRESSING COMPATIBILITY

Programs can be easily transported from one member of the M68000 Family to another in an upward compatible fashion. The user object code of each early member of the family is upward compatible with newer members, and can be executed on the newer microprocessor without change. The address extension word(s) are encoded with the information that allows the MC68020 to distinguish the new address extensions to the basic M68000 Family architecture. The address extension words for the early MC68000, MC68008, and MC68010 microprocessors and for the newer 32-bit MC68020 microprocessors are shown in Figure 2-14. Notice the encoding for SCALE used by the MC68020 is a compatible extension of the M68000 architecture. A value of zero for SCALE is the same encoding for both extension words; hence software that uses this encoding is both upward and downward compatible across all processors in the product line. However, the other values of SCALE are not found in both extension formats; so, while software can be easily migrated in an upward compatible direction, only nonscaled addressing is supported in a downward fashion. If the MC68000 were to execute an instruction that encoded a scaling factor, the scaling factor would be ignored and not access the desired memory address. The earlier microprocessors have no knowledge of the extension word formats implemented by newer processors, and while they do detect illegal instructions, they do not decode invalid encodings of the extension words as exceptions.

2.8 OTHER DATA STRUCTURES

Stacks and queues are widely used data structures. The MC68020 implements a system stack and also provides instructions that support the use of user stacks and queues.

MC68000/MC68008/MC68010 Address Extension Word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	REGISTER			W/L	0	0	0	DISPLACEMENT INTEGER							

D/A: 0 = Data Register Select
1 = Address Register Select
W/L: 0 = Word-Sized Operation
1 = Long-Word-Sized Operation

MC68020 Address Extension Word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	REGISTER			W/L	SCALE		0	DISPLACEMENT INTEGER							

D/A: 0 = Data Register Select
1 = Address Register Select
W/L: 0 = Word-Sized Operation
1 = Long-Word-Sized Operation
SCALE: 00 = Scale Factor 1 (Compatible with MC68000)
01 = Scale Factor 2 (Extension to MC68000)
10 = Scale Factor 4 (Extension to MC68000)
11 = Scale Factor 8 (Extension to MC68000)

Figure 2-14. M68000 Family Address Extension Words

2.8.1 System Stack

Address register seven (A7) is used as the system stack pointer (SP). Any of the three system stack registers is active at any one time. The M and S bits of the status register determine which stack pointer is used. When S = 0 indicating user mode (user privilege level), the user stack pointer (USP) is the active system stack pointer and the master and interrupt stack pointers cannot be referenced. When S = 1 indicating supervisor mode (at supervisor privilege level) and M = 1, the master stack pointer (MSP) is the active system stack pointer. When S = 1 and M = 0, the interrupt stack pointer (ISP) is the active system stack pointer. This mode is the MC68020 default mode after reset and corresponds to the MC68000, MC68008, and MC68010 supervisor mode. The term supervisor stack pointer (SSP) refers to the master or interrupt stack pointers, depending on the state of the M bit. When M = 1, the term SSP (or A7) refers to the MSP address register. When M = 0, the term SSP (or A7) refers to the ISP address register. The active system stack pointer is implicitly referenced by all instructions that use the system stack. Each system stack fills from high to low memory.

A subroutine call saves the program counter on the active system stack, and the return restores it from the active system stack. During the processing of traps and interrupts, both the program counter and the status register are saved on the supervisor stack (either master or interrupt). Thus, the execution of supervisor level code is independent of user code and the condition of the user stack; conversely, user programs use the user stack pointer independently of supervisor stack requirements.

In order to keep data on the system stack aligned for maximum efficiency, the active stack pointer is automatically decremented or incremented by two for all byte-size operands moved to or from the stack. In long-word-organized memory, aligning the stack pointer on a long word address significantly increases the efficiency of stacking exception frames, subroutine calls and returns, and other stacking operations.

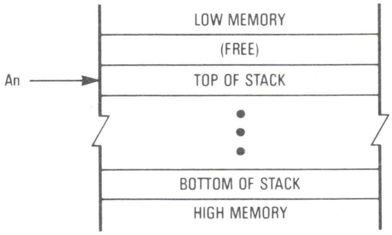
2.8.2 User Program Stacks

The user can implement stacks with the address register indirect with postincrement and predecrement addressing modes. With address register An (n = 6–0), the user can implement a stack that is filled either from high memory to low memory or from low memory to high. Important considerations are:

- Use the predecrement mode to decrement the register before its contents are used as the pointer to the stack.
- Use the postincrement mode to increment the register after its contents are used as the pointer to the stack.
- Maintain the stack pointer correctly when byte, word, and long word items are mixed in these stacks.

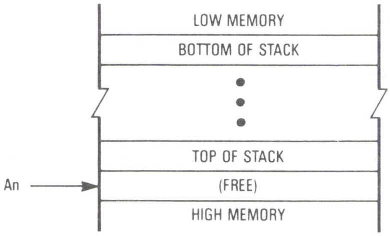
To implement stack growth from high to low memory, use:
 − (An) to push data on the stack,
(An) + to pull data from the stack.

For this type of stack, after either a push or a pull operation, register An points to the top item on the stack. This is illustrated as:



To implement stack growth from low to high memory, use:
(An)+ to push data on the stack,
−(An) to pull data from the stack.

In this case, after either a push or pull operation, register An points to the next available space on the stack. This is illustrated as:.

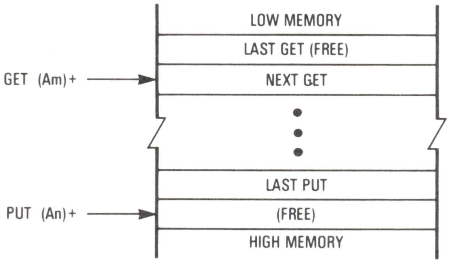


2.8.3 Queues

The user can implement queues with the address register indirect with postincrement or predecrement addressing modes. Using a pair of address registers (two of A6–A0), the user can implement a queue which is filled either from high memory to low memory, or from low memory to high. Two registers are used because queues are pushed from one end and pulled from the other. One register, An, contains the "put" pointer; the other, Am, the "get" pointer.

To implement growth of the queue from low to high memory, use:
(An)+ to put data into the queue,
(Am)+ to get data from the queue.

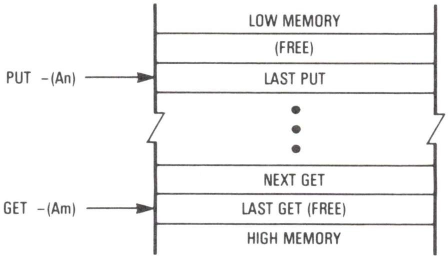
After a "put" operation, the "put" address register points to the next available space in the queue, and the unchanged "get" address register points to the next item to be removed from the queue. After a "get" operation, the "get" address register points to the next item to be removed from the queue, and the unchanged "put" address register points to the next available space in the queue. This is illustrated as:


LOW MEMORY
LAST GET (FREE)
GET (Am)+
NEXT GET
LAST PUT
PUT (An)+
(FREE)
HIGH MEMORY

To implement the queue as a circular buffer, the relevant address register should be checked and adjusted, if necessary, before performing the "put" or "get" operation. The address register is adjusted by subtracting the buffer length (in bytes) from the register.

To implement growth of the queue from high to low memory, use:
- −(An) to put data into the queue,
- −(Am) to get data from the queue.

After a "put" operation, the "put" address register points to the last item placed in the queue, and the unchanged "get" address register points to the last item removed from the queue. After a "get" operation, the "get" address register points to the last item removed from the queue, and the unchanged "put" address register points to the last item placed in the queue. This is illustrated as:


LOW MEMORY
(FREE)
PUT −(An)
LAST PUT
NEXT GET
GET −(Am)
LAST GET (FREE)
HIGH MEMORY

To implement the queue as a circular buffer, the "get" or "put" operation should be performed first, and then the relevant address register should be checked and adjusted, if necessary. The address register is adjusted by adding the buffer length (in bytes) to the register contents.

SECTION 3
INSTRUCTION SET

This section describes the set of instructions provided in the MC68020 microprocessor, and shows how to use them. It includes descriptions of the instruction format and the operands used by instructions. After a summary of the instructions by category, the section provides a detailed description of the operation of each instruction, in alphabetical order. Programming information for specific instructions is next, followed by a description of condition code computation and an instruction format summary.

3.1 INSTRUCTION FORMAT

All instructions consist of at least one word; some have as many as 11 words as shown in Figure 3-1. The first word of the instruction, called the operation word, specifies the length of the instruction and the operation to be performed. The remaining words, called extension words, further specify the instruction and operands. These words may be immediate operands, extensions to the effective address mode specified in the operation word, branch displacements, bit number or bit field specifications, special register specifications, trap operands, pack/unpack constants, argument counts, or coprocessor condition codes.

Besides the operation code, which specifies the function to be performed, an instruction defines the location of every operand for the function. Instructions specify an operand location in one of three ways:

- Register Specification — A register field of the instruction contains the number of the register.
- Effective Address — An effective address field of the instruction contains address mode information.
- Implicit Reference — The definition of an instruction implies the use of specific registers.

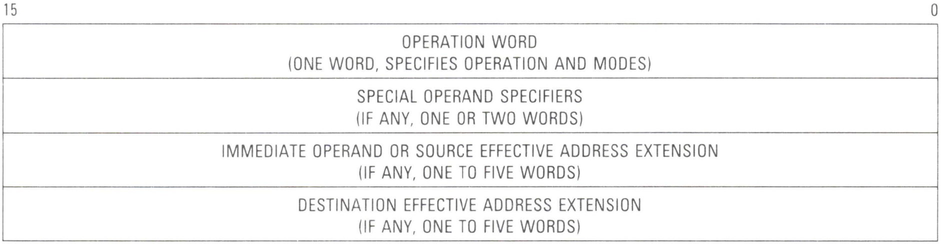

15
0
OPERATION WORD
(ONE WORD, SPECIFIES OPERATION AND MODES)
SPECIAL OPERAND SPECIFIERS
(IF ANY, ONE OR TWO WORDS)
IMMEDIATE OPERAND OR SOURCE EFFECTIVE ADDRESS EXTENSION
(IF ANY, ONE TO FIVE WORDS)
DESTINATION EFFECTIVE ADDRESS EXTENSION
(IF ANY, ONE TO FIVE WORDS)

Figure 3-1. Instruction Word General Format

The register field within an instruction specifies the register to be used. Other fields within the instruction specify whether the register selected is an address or data register and how the register is to be used. Section 2 contains detailed register information.

Effective address information includes the registers, displacements, and absolute addresses for the effective address modes. Section 2 describes the effective address modes in detail in **SECTION 2 DATA ORGANIZATION AND ADDRESSING CAPABILITIES**.

Certain instructions are operations for specific registers. These instructions imply the required registers.

3.2 INSTRUCTION SUMMARY

The instructions form a set of tools to perform the following operations:

- Data Movement
- Integer Arithmetic
- Logical
- Shift and Rotate
- Bit Manipulation
- Bit Field Manipulation
- Binary Coded Decimal Arithmetic
- Program Control
- System Control
- Multiprocessor Communications

The complete range of instruction capabilities combined with the addressing modes described previously provide flexibility for program development.

The following notations are used in this Section. In the operand syntax statements of the instruction descriptions, the operand on the right is the destination operand.

An = any address register, A7–A0
Dn = any data register, D7–D0
Rn = any address or data register
CCR = condition code register (lower byte of status register)
cc = condition codes from CCR
SR = status register
cpcc = coprocessor condition code
SP = active stack pointer
USP = user stack pointer
ISP = supervisor/interrupt stack pointer
MSP = supervisor/master stack pointer
SSP = supervisor (master or interrupt) stack pointer
DFC = destination function code register
SFC = source function code register
Rc = control register (VBR, SFC, DFC, CACR, CAAR)
d = displacement; d_{16} is a 16-bit displacement
⟨ea⟩ = effective address
list = list of registers, for example, D3–D0
#⟨data⟩ = immediate data; a literal integer
{offset:width} = bit field selection
label = assembly program label
[7] = bit 7 of an operand
[31:24] = bits 31 through 24 of operand (high-order byte of a register)

X = extend (X) bit in CCR
N = negative (N) bit in CCR
Z = zero (Z) bit in CCR
V = overflow (V) bit in CCR
C = carry (C) bit in CCR
+ = arithmetic addition or postincrement indicator
− = arithmetic subtraction or predecrement indicator
* = arithmetic multiplication
/ = arithmetic division or conjunction symbol
~ = invert; operand is logically complemented
Λ = logical AND
V = logical OR
⊕ = logical exclusive OR
Dc = data register, D7–D0 used during compare
Du = data register, D7–D0 used during update
Dr, Dq = data registers, remainder or quotient of divide
Dh, Dl = data registers, high or low order 32 bits of product
MSW = most significant word
LSW = least significant word
MSB = most significant bit
FC = function code
{R/W} = read or write indicator
[An] = address extensions

3.2.1 Data Movement Instructions

The MOVE instruction with its associated addressing modes is the basic means of transferring and storing addresses and data. MOVE instructions transfer byte, word, and long word operands from memory to memory, memory to register, register to memory, and register to register. Address movement instructions (MOVE or MOVEA) transfer word and long word operands and ensure that only valid address manipulations are executed. In addition to the general MOVE instructions there are several special data movement instructions: move multiple registers (MOVEM), move peripheral data (MOVEP), move quick (MOVEQ), exchange registers (EXG), load effective address (LEA), push effective address (PEA), link stack (LINK), and unlink stack (UNLK). Table 3-1 is a summary of the data movement operations.

3.2.2 Integer Arithmetic Instructions

The arithmetic operations include the four basic operations of add (ADD), subtract (SUB), multiply (MUL), and divide (DIV) as well as arithmetic compare (CMP, CMPM, CMP2), clear (CLR), and negate (NEG). The instruction set includes ADD, CMP, and SUB instructions for both address and data operations with all operand sizes valid for data operations. Address operands consist of 16 or 32 bits. The clear and negate instructions apply to all sizes of data operands.

Table 3-1. Data Movement Operations

Instruction	Operand Syntax	Operand Size	Operation
EXG	Rn, Rn	32	Rn ↔ Rn
LEA	⟨ea⟩,An	32	⟨ea⟩ → An
LINK	An,#⟨d⟩	16, 32	Sp-4 → SP; An → (SP); SP → An; SP + d → SP
MOVE MOVEA	⟨ea⟩,⟨ea⟩ ⟨ea⟩,An	8, 16, 32 16,32 → 32	source → destination
MOVEM	list,⟨ea⟩ ⟨ea⟩,list	16, 32 16, 32 → 32	listed registers → destination source → listed registers
MOVEP	Dn, (d_{16},An) (d_{16},An),Dn	16,32	Dn[31:24] → (An + d); Dn[23:16] → (An + d + 2); Dn[15:8] → (An + d + 4); Dn[7:0] → (An + d + 6) (An + d) → Dn[31:24]; (An + d + 2) → Dn [23:16]; (An + d + 4) → Dn[15:8]; (An + d + 6) → Dn[7:0]
MOVEQ	#⟨data⟩,Dn	8 → 32	immediate data → destination
PEA	⟨ea⟩	32	SP-4 → SP; ⟨ea⟩ → (SP)
UNLK	An	32	An → SP; (SP) → An; SP + 4 → SP

Signed and unsigned MUL and DIV instructions include:

- Word multiply to produce a long word product
- Long word multiply to produce a long word or quad word product
- Division of a long word dividend by a word divisor (word quotient and word remainder)
- Division of a long word or quad word dividend by a long word divisor (long word quotient and long word remainder)

A set of extended instructions provides multiprecision and mixed size arithmetic. These instructions are: add extended (ADDX), subtract extended (SUBX), sign extend (EXT), and negate binary with extend (NEGX). Refer to Table 3-2 for a summary of the integer arithmetic operations.

3.2.3 Logical Instructions

The logical operation instructions (AND, OR, EOR, and NOT) perform logical operations with all sizes of integer data operands. A similar set of immediate instructions (ANDI, ORI, and EORI) provide these logical operations with all sizes of immediate data. The TST instruction compares the operand with zero arithmetically, placing the result in the condition code register.

Table 3-3 summarizes the logical operations.

3.2.4 Shift and Rotate Instructions

The arithmetic shift instructions ASR and ASL and logical shift instructions LSR and LSL provide shift operations in both directions. The ROR, ROL, ROXR, and ROXL instructions

Table 3-2. Integer Arithmetic Operations

Instruction	Operand Syntax	Operand Size	Operation
ADD ADDA	Dn,⟨ea⟩ ⟨ea⟩,Dn ⟨ea⟩,An	8, 16, 32 8, 16, 32 16, 32	source + destination ➧ destination
ADDI ADDQ	#⟨data⟩,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 16, 32 8, 16, 32	immediate data + destination ➧ destination
ADDX	Dn,Dn – (An), – (An)	8, 16, 32 8, 16, 32	source + destination + X ➧ destination
CLR	⟨ea⟩	8, 16, 32	0 ➧ destination
CMP CMPA	⟨ea⟩,Dn ⟨ea⟩,An	8, 16, 32 16, 32	destination – source
CMPI	#⟨data⟩,⟨ea⟩	8, 16, 32	destination – immediate data
CMPM	(An) + ,(An) +	8, 16, 32	destination – source
CMP2	⟨ea⟩,Rn	8, 16, 32	lower bound ⟨ = Rn ⟨ = upper bound
DIVS DIVU DIVSL DIVUL	⟨ea⟩,Dn ⟨ea⟩,Dr:Dq ⟨ea⟩,Dq ⟨ea⟩,Dr:Dq	32 16 ➧ 16:16 64 32 ➧ 32:32 32 32 ➧ 32 32 32 ➧ 32:32	destination source ➧ destination (signed or unsigned)
EXT EXTB	Dn Dn Dn	8 ➧ 16 16 ➧ 32 8 ➧ 32	sign extended destination ➧ destination
MULS MULU	⟨ea⟩,Dn ⟨ea⟩,Dl ⟨ea⟩,Dh:Dl	16 × 16 ➧ 32 32 × 32 ➧ 32 32 × 32 ➧ 64	source*destination ➧ destination (signed or unsigned)
NEG	⟨ea⟩	8, 16, 32	0 – destination ➧ destination
NEGX	⟨ea⟩	8, 16, 32	0 – destination – X ➧ destination
SUB SUBA	⟨ea⟩,Dn Dn,⟨ea⟩ ⟨ea⟩,An	8, 16, 32 8, 16, 32 16, 32	destination = source ➧ destination
SUBI SUBQ	#⟨data⟩,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 16, 32 8, 16, 32	destination – immediate data ➧ destination
SUBX	Dn,Dn – (An), – (An)	8, 16, 32 8, 16, 32	destination – source – X ➧ destination

Table 3-3. Logical Operations

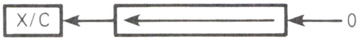
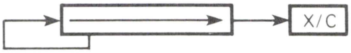
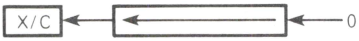
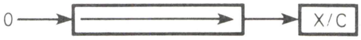
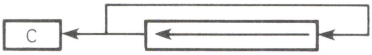
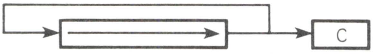
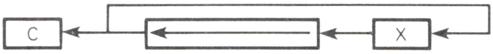
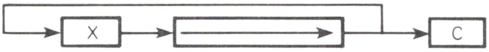
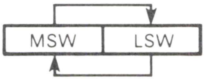
Instruction	Operand Syntax	Operand Size	Operation
AND	⟨ea⟩,Dn Dn,⟨ea⟩	8, 16, 32 8, 16, 32	source \ destination ➧ destination
ANDI	#⟨data⟩,⟨ea⟩	8, 16, 32	immediate data \ destination ➧ destination
EOR	Dn,⟨ea⟩	8, 16, 32	source ⊕ destination ➧ destination
EORI	#⟨data⟩,⟨ea⟩	8, 16, 32	immediate data ⊕ destination ➧ destination
NOT	⟨ea⟩	8, 16, 32	~ destination ➧ destination
OR	⟨ea⟩,Dn Dn,⟨ea⟩	8, 16, 32 8, 16, 32	source V destination ➧ destination
ORI	#⟨data⟩,⟨ea⟩	8, 16, 32	immediate data V destination ➧ destination
TST	⟨ea⟩	8, 16, 32	source – 0 to set condition codes

perform rotate (circular shift) operations, with and without the extend bit. All shift and rotate operations can be performed on either registers or memory.

Register shift and rotate operations shift all operand sizes. The shift count may be specified in the instruction operation word (to shift from 1-8 places) or in a register (modulo 64 shift count).

Memory shift and rotate operations shift word-length operands one bit position only. The SWAP instruction exchanges the 16-bit halves of a register. Performance of shift/rotate instructions is enhanced so that use of the ROR and ROL instructions with a shift count of eight allows fast byte swapping. Table 3-4 is a summary of the shift and rotate operations.

Table 3-4. Shift and Rotate Operations

Instruction	Operand Syntax	Operand Size	Operation
ASL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ASR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
LSL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
LSR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROXL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROXR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
SWAP	Dn	32	

3.2.5 Bit Manipulation Instructions

Bit manipulation operations are accomplished using the following instructions: bit test (BTST), bit test and set (BSET), bit test and clear (BCLR), and bit test and change (BCHG). All bit manipulation operations can be performed on either registers or memory. The bit number is specified as immediate data or in a data register. Register operands are 32 bits long, and memory operands are 8 bits long. In Table 3-5, the summary of the bit manipulation operations, Z refers to bit 2, the zero bit of the status register.

Table 3-5. Bit Manipulation Operations

Instruction	Operand Syntax	Operand Size	Operation
BCHG	Dn,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 32 8, 32	~ (⟨bit number⟩ of destination) → Z → bit of destination
BCLR	Dn,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 32 8, 32	~ (⟨bit number⟩ of destination) → Z; 0 → bit of destination
BSET	Dn,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 32 8, 32	~ (⟨bit number⟩ of destination) → Z; 1 → bit of destination
BTST	Dn,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 32 8, 32	~ (⟨bit number⟩ of destination) → Z

3.2.6 Bit Field Instructions

The MC68020 supports variable length bit field operations on fields of up to 32 bits. The bit field insert (BFINS) instruction inserts a value into a bit field. Bit field extract unsigned (BFEXTU) and bit field extract signed (BFEXTS) extract a value from the field. Bit field find first one (BFFFO) finds the first bit that is set in a bit field. Also included are instructions that are analogous to the bit manipulation operations; bit field test (BFTST), bit field test and set (BFSET), bit field test and clear (BFCLR), and bit field test and change (BFCHG). Table 3-6 is a summary of the bit field operations.

Table 3-6. Bit Field Operations

Instruction	Operand Syntax	Operand Size	Operation
BFCHG	⟨ea⟩ {offset:width}	1-32	Field → Field
BFCLR	⟨ea⟩ {offset:width}	1-32	0's → Field
BFEXTS	⟨ea⟩ {offset:width},Dn	1-32	Field → Dn; Sign Extended
BFEXTU	⟨ea⟩ {offset:width},Dn	1-32	Field → Dn; Zero Extended
BFFFO	⟨ea⟩ {offset:width},Dn	1-32	Scan for first bit set in Field; offset → Dn
BFINS	Dn,⟨ea⟩ {offset:width}	1-32	Dn → Field
BFSET	⟨ea⟩ {offset:width}	1-32	1's → Field
BFTST	⟨ea⟩ {offset:width}	1-32	Field MSB → N; (OR of all bits in field) → Z

NOTE: All bit field instructions set the N and Z bits as shown for BFTST before performing the specified operation.

3.2.7 Binary Coded Decimal Instructions

Five instructions support operations on binary coded decimal numbers. The arithmetic operations on packed binary coded decimal numbers are: add decimal with extend (ABCD), subtract decimal with extend (SBCD), and negate decimal with extend (NBCD). PACK and UNPACK instructions aid in the conversion of byte encoded numeric data, such as ASCII or EBCDIC strings, to BCD data and vice versa. Table 3-7 is a summary of the binary coded decimal operations.

Table 3-7. Binary Coded Decimal Operations

Instruction	Operand Syntax	Operand Size	Operation
ABCD	Dn,Dn – (An), – (An)	8 8	source$_{10}$ + destination$_{10}$ + X ➧ destination
NBCD	⟨ea⟩	8	0 # destination$_{10}$ – X ➧ destination
PACK	– (An), – (An) #⟨data⟩ Dn,Dn,#⟨data⟩	16 ➧ 8 16 ➧ 8	unpackaged source + immediate data ➧ packed destination
SBCD	Dn,Dn – (An), – (An)	8 8	destination$_{10}$ – source$_{10}$ – X ➧ destination
UNPK	– (An), – (An) #⟨data⟩ Dn,Dn,#⟨data⟩	8 ➧ 16 8 ➧ 16	packed source ➧ unpacked source unpacked source + immediate data ➧ unpacked destination

3.2.8 Program Control Instructions

A set of subroutine call and return instructions and conditional and unconditional branch instructions perform program control operations. Table 3-8 summarizes these instructions.

Table 3-8. Program Control Operations (Sheet 1 of 2)

Instruction	Operand Syntax	Operand Size	Operation
CONDITIONAL			
Bcc	⟨label⟩	8, 16, 32	if condition true, then PC + d ➧ PC
DBcc	Dn,⟨label⟩	16	if condition false, then Dn – 1 ➧ Dn if Dn ≠ – 1, then PC + d ➧ PC
Scc	⟨ea⟩	8	if condition true, then 1's ➧ destination; else 0's ➧ destination
UNCONDITIONAL			
BRA	⟨label⟩	8, 16, 32	PC + d ➧ PC
BSR	⟨label⟩	8, 16, 32	SP – 4 ➧ SP; PC ➧ (SP); PC + d ➧ PC
CALLM	#<data>, <ea>	none	Save module state in stack frame; load new module state from destination
JMP	⟨ea⟩	none	destination ➧ PC
JSR	⟨ea⟩	none	SP – 4 ➧ SP; PC ➧ (SP); destination ➧ PC
NOP	none	none	PC + 2 ➧ PC

Table 3-8. Program Control Operations (Sheet 2 of 2)

Instruction	Operand Syntax	Operand Size	Operation
			RETURNS
RTD	#⟨d⟩	16	(SP) → PC; SP + 4 + d → SP
RTM	Rn	none	Reload saved module state from stack frame; place module data area pointer in Rn
RTR	none	none	(SP) → CCR; SP + 2 → SP; (SP) → PC; SP + 4 → SP
RTS	none	none	(SP) → PC; SP + 4 → SP

Letters cc in the instruction mnemonic opcodes specify testing one of the following condition codes:

CC — Carry clear
CS — Carry set
EQ — Equal
F — Never true*
GE — Greater or equal
GT — Greater than
HI — High
LE — Less or equal
LS — Low or same
LT — Less than
MI — Minus
NE — Not equal
PL — Plus
T — Always true*
VC — Overflow clear
VS — Overflow set

* Not applicable to the Bcc or cpBcc instructions.

3.2.9 System Control Instructions

Privileged instructions, trapping instructions, and instructions that use or modify the condition code register provide system control operations. Table 3-9 summarizes these instructions. The preceding list of condition code representations applies to the TRAPcc instruction. All of these instructions cause the processor to flush the instruction pipe.

Table 3-9 System Control Operations (Sheet 1 of 2)

Instruction	Operand Syntax	Operand Size	Operation
			PRIVILEGED
ANDI	#⟨data⟩,SR	16	immediate data \ SR → SR
EORI	#⟨data⟩,SR	16	immediate data ⊕ SR → SR
MOVE	⟨ea⟩,SR SR,⟨ea⟩	16 16	source → SR SR → destination
MOVE	USP,An An,USP	32 32	USP → An An → USP
MOVEC	Rc,Rn Rn,Rc	32 32	Rc → Rn Rn → Rc
MOVES	Rn,⟨ea⟩ ⟨ea⟩,Rn	8, 16, 32	Rn → destination using DFC source using SFC → Rn
ORI	#⟨data⟩,SR	16	immediate data V SR → SR
RESET	none	none	assert $\overline{\text{RESET}}$ line

Table 3-9 System Control Operations (Sheet 2 of 2)

Instruction	Operand Syntax	Operand Size	Operation
PRIVILEGED			
RTE	none	none	(SP) ▶ SR; SP+2 ▶ SP; (SP) ▶ PC; SP+4 ▶ SP; Restore stack according to format
STOP	#⟨data⟩	16	immediate data ▶ SR; STOP
TRAP GENERATING			
BKPT	#⟨data⟩	none	if breakpoint cycle acknowledged, then execute returned operation word, else trap as illegal instruction
CHK	⟨ea⟩,Dn	16, 32	if Dn <0 or Dn>(ea), then CHK exception
CHK2	⟨ea⟩,Rn	8, 16, 32	if Rn <lower bound or Rn>upper bound, then CHK exception
ILLEGAL	none	none	SSP −2 ▶ SSP; Vector Offset ▶ (SSP); SSP −4 ▶ SSP; PC ▶ (SSP); SSP −2 ▶ SSP; SR ▶ (SSP); Illegal Instruction Vector Address ▶ PC
TRAP	#⟨data⟩	none	SSP −2 ▶ SSP; Format and Vector Offset ▶ (SSP) SSP −4 ▶ SSP; PC ▶ (SSP); SSP −2 ▶ SSP; SR ▶ (SSP); Vector Address ▶ PC
TRAP GENERATING (CONTINUED)			
TRAPcc	none #⟨data⟩	none 16, 32	if cc true, then TRAP exception
TRAPV	none	none	if V then take overflow TRAP exception
CONDITION CODE REGISTER			
ANDI	#⟨data⟩,CCR	8	immediate data \ CCR ▶ CCR
EORI	#⟨data⟩,CCR	8	immediate data ⊕ CCR ▶ CCR
MOVE	⟨ea⟩,CCR CCR,⟨ea⟩	16 16	source ▶ CCR CCR ▶ destination
ORI	#⟨data⟩,CCR	8	immediate data V CCR ▶ CCR

3.2.10 Multiprocessor Instructions

The TAS, CAS, and CAS2 instructions coordinate the operations of processors in multiprocessing systems. These instructions use read-modify-write bus cycles to ensure uninterrupted updating of memory. Coprocessor instructions control the coprocessor operations. Table 3-10 summarizes these instructions.

3.3 INSTRUCTION SET DETAILS

This paragraph contains detailed information about each instruction in the MC68020 instruction set. First, it describes the notation and the format of the instruction description. Then each instruction is described in detail. The instruction descriptions are arranged in alphabetical order by instruction mnemonic.

Table 3-10. Multiprocessor Operations

Instruction	Operand Syntax	Operand Size	Operation
READ-MODIFY-WRITE			
CAS	Dc,Du,⟨ea⟩	8, 16, 32	destination − Dc ▶ CC; if Z then Du ▶ destination else destination ▶ Dc
CAS2	Dc1:Dc2, Du1:Du2, (Rn):(Rn)	16, 32	dual operand CAS
TAS	⟨ea⟩	8	destination − 0; set condition codes; 1 ▶ destination [7]
COPROCESSOR			
cpBcc	⟨label⟩	16, 32	if cpcc true then PC+d ▶ PC
cpDBcc	⟨label⟩,Dn	16	if cpcc false then Dn − 1 ▶ Dn if Dn ≠ − 1, then PC+d ▶ PC
cpGEN	User Defined	User Defined	operand ▶ coprocessor
cpRESTORE	⟨ea⟩	none	restore coprocessor state from ⟨ea⟩
cpSAVE	⟨ea⟩	none	save coprocessor state at ⟨ea⟩
cpScc	⟨ea⟩	8	if cpcc true, then 1's ▶ destination; else 0's ▶ destination
cpTRAPcc	none #⟨data⟩	none 16, 32	if cpcc true then TRAPcc exception

3.3.1 Notation and Format

The instruction descriptions use notational conventions for the operands, the subfields and qualifiers, and the operations performed by the instructions. In the syntax descriptions, the left operand is the source operand, and the right operand is the destination operand. The notational conventions listed in **SECTION 3.2 INSTRUCTION SUMMARY** apply. The following lists contain the additional notations used in the instruction descriptions.

Notation for operands:

PC—Program counter
SR—Status register
V—Overflow condition code
Immediate Data—Immediate data from the instruction
Source—Source contents
Destination—Destination contents
Vector—Location of exception vector

By convention, the destination operand is the operand on the right.

Notation for subfields and qualifiers:

⟨bit⟩ of ⟨operand⟩—Selects a single bit of the operand
⟨ea⟩{offset:width}—Selects a bit field
(⟨operand⟩)—The contents of the referenced location
⟨operand⟩$_{10}$—The operand is binary coded decimal, operations are performed in decimal

(⟨address register⟩)—The register indirect operator which indicates that the operand register points to the memory location of the instruction operand. The optional mode qualifiers are −, +, (d) and (d,ix)
−(⟨address register⟩)
(⟨address register⟩)+

#xxx or #⟨data⟩—Immediate data that follows the instruction word(s)

Notations for operations that have two operands, written ⟨operand⟩ ⟨op⟩ ⟨operand⟩, where ⟨op⟩ is one of the following:

▶—The source operand is moved to the destination operand
◀▶—The two operands are exchanged
+—The operands are added
−—The destination operand is subtracted from the source operand
*—The operands are multiplied
/—The source operand is divided by the destination operand
⟨—Relational test, true if source operand is less than destination operand
⟩—Relational test, true if source operand is greater than destination operand
shifted by—The source operand is shifted or rotated by the number of positions specified by the second operand
rotated by

Notation for single-operand operations:

~⟨operand⟩—The operand is logically complemented
⟨operand⟩sign-extended—The operand is sign extended, all bits of the upper portion are made equal to the high order bit of the lower portion
⟨operand⟩tested—The operand is compared to 0 and the condition codes are set appropriately

Notation for other operations:

TRAP—Equivalent to Format/Offset Word ▶ (SSP); SSP-2 ▶ SSP; PC ▶ (SSP); SSP-4 ▶ SSP; SR ▶ (SSP); SSP-2 ▶ SSP; (vector) ▶ PC
STOP—Enter the stopped state, waiting for interrupts
If ⟨condition⟩ then ⟨operations⟩ else ⟨operations⟩—The condition is tested. If true, the operations after "then" are performed. If the condition is false and the optional "else" clause is present, the operations after "else" are performed. If the condition is false and else is omitted, the instruction performs no operation. Refer to the description of Bcc instruction as an example.

3.3.2 Condition Code Register

The condition code register portion of the status register contains five bits:

X — Extend
N — Negative
Z — Zero
V — Overflow
C — Carry

The last four bits represent a condition of the result of a processor operation. Table 3-10 lists the effect of each instruction on these bits. The X bit is an operand for multiprecision computations; when it is used, it is set to the value of the carry bit. The carry bit and the multiprecision extend bit are separate in the M68000 Family to simplify programming techniques that use them. Refer to Table 3-4 as an example.

Program and system control instructions use certain combinations of these bits to control program and system flow. Table 3-11 lists the combinations of these bits and their interpretations.

In the instruction set descriptions, the condition code register is shown as follows:

X	N	Z	V	C

where:

X (extend)

Set to the value of the C bit for many arithmetic operations. Otherwise not affected or set to a specified result.

N (negative)

Set if the most significant bit of the result is set. Cleared otherwise.

Z (zero)

Set if the result equals zero. Cleared otherwise.

V (overflow)

Set if arithmetic overflow occurs. This implies that the result cannot be represented in the operand size. Cleared otherwise.

C (carry)

Set if a carry out of the most significant bit of the operands occurs, for an addition. Also, set if a borrow occurs in a subtraction. Cleared otherwise.

The following symbols are shown in the square representing each condition code:

* = Set according to the result of the operation
— = Not affected by the operation
0 = Cleared
1 = Set
U = Undefined after the operation

3.3.3 Instruction Descriptions

Figure 3-2 shows the format of the instruction descriptions. The attributes line specifies the size of the operands of an instruction. When an instruction can use operands of more than one size, a suffix is used with the mnemonic of the instruction:

.B—Byte operands
.W—Word operands
.L—Long operands

Table 3-11. Condition Code Computations (Sheet 1 of 2)

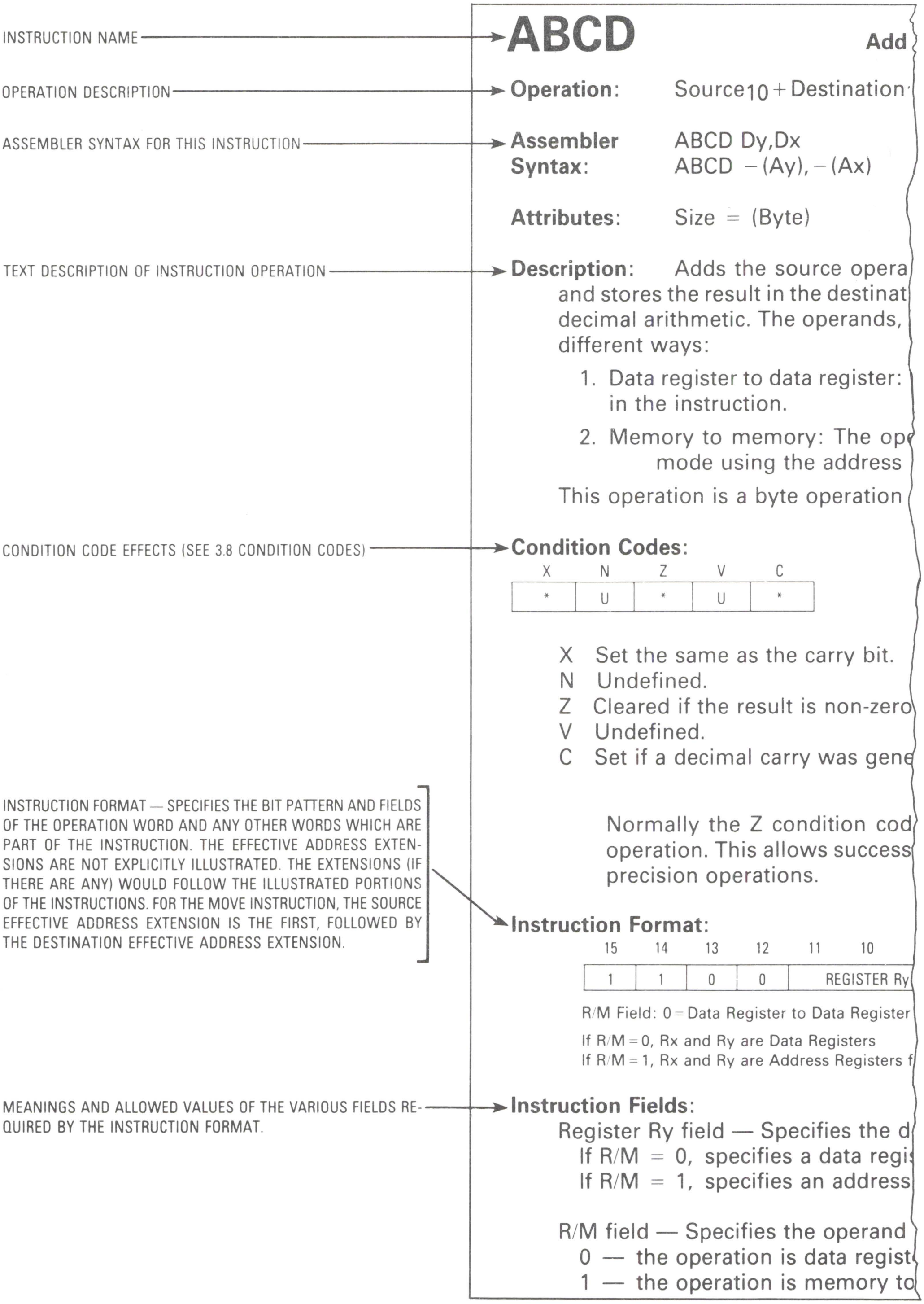
Operations	X	N	Z	V	C	Special Definition
ABCD	*	U	?	U	?	C = Decimal Carry $Z = Z \backslash \overline{Rm} \backslash \ldots \backslash \overline{R0}$
ADD, ADDI, ADDQ	*	*	*	?	?	$V = Sm \backslash Dm \backslash \overline{Rm} V \overline{Sm} \backslash \overline{Dm} \backslash Rm$ $C = Sm \backslash Dm V \overline{Rm} \backslash Dm V Sm \backslash \overline{Rm}$
ADDX	*	*	?	?	?	$V = Sm \backslash Dm \backslash \overline{Rm} V \overline{Sm} \backslash \overline{Dm} \backslash Rm$ $C = Sm \backslash Dm V \overline{Rm} \backslash Dm V Sm \backslash \overline{Rm}$ $Z = Z \backslash \overline{Rm} \backslash \ldots \backslash \overline{R0}$
AND, ANDI, EOR, EORI, MOVEQ, MOVE, OR, ORI, CLR, EXT, NOT, TAS, TST	—	*	*	0	0	
CHK	—	*	U	U	U	
CHK2, CMP2	—	U	?	U	?	$Z = (R = LB) V (R = UB)$ $C = (LB < = UB) \backslash (IR < LB) V (R > UB)) V (UB < LB) \backslash (R > UB) \backslash (R < LB)$
SUB, SUBI, SUBQ	*	*	*	?	?	$V = \overline{Sm} \backslash Dm \backslash \overline{Rm} V Sm \backslash \overline{Dm} \backslash Rm$ $C = Sm \backslash \overline{Dm} V Rm \backslash \overline{Dm} V Sm \backslash Rm$
SUBX	*	*	?	?	?	$V = \overline{Sm} \backslash Dm \backslash \overline{Rm} V Sm \backslash \overline{Dm} \backslash Rm$ $C = Sm \backslash \overline{Dm} V Rm \backslash \overline{Dm} V Sm \backslash Rm$ $Z = Z \backslash \overline{Rm} \backslash \ldots \backslash \overline{R0}$
CAS, CAS2, CMP, CMPI, CMPM	—	*	*	?	?	$V = \overline{Sm} \backslash Dm \backslash \overline{Rm} V Sm \backslash \overline{Dm} \backslash Rm$ $C = Sm \backslash \overline{Dm} V Rm \backslash \overline{Dm} V Sm \backslash Rm$
DIVS, DUVI	—	*	*	?	0	V = Division Overflow
MULS, MULU	—	*	*	?	0	V = Multiplication Overflow
SBCD, NBCD	*	U	?	U	?	C = Decimal Borrow $Z = Z \backslash \overline{Rm} \backslash \ldots \backslash \overline{Ro}$
NEG	*	*	*	?	?	$V = Dm \backslash Rm$ $C = Dm V Rm$
NEGX	*	*	?	?	?	$V = Dm \backslash Rm$ $C = Dm V Rm$ $Z = Z \backslash \overline{Rm} \backslash \ldots \backslash \overline{R0}$
BTST, BCHG, BSET, BCLR	—	—	?	—	—	$Z = \overline{Dn}$
BFTST, BFCHG, BFSET, BFCLR	—	?	?	0	0	$N = Dm$ $Z = \overline{Dm} \backslash \overline{DM - 1} \backslash \ldots \backslash \overline{D0}$
BFEXTS, BFEXTU, BFFFO	—	?	?	0	0	$N = Sm$ $Z = \overline{Sm} \backslash \overline{Sm - 1} \backslash \ldots \backslash \overline{S0}$
BFINS	—	?	?	0	0	$N = Dm$ $Z = \overline{Dm} \backslash \overline{DM - 1} \backslash \ldots \backslash \overline{D0}$
ASL	*	*	*	?	?	$V = Dm \backslash (\overline{Dm - 1} V \ldots V \overline{Dm - r}) V \overline{Dm} \backslash (DM - 1 V \ldots + Dm - r)$ $C = \overline{Dm - r + 1}$
ASL (R = 0)	—	*	*	0	0	
LSL, ROXL	*	*	*	0	?	$C = Dm - r + 1$
LSR (r = 0)	—	*	*	0	0	
ROXL (r = 0)	—	*	*	0	?	C = X
ROL	—	*	*	0	?	$C = Dm - r + 1$
ROL (r = 0)	—	*	*	0	0	

Table 3-11. Condition Code Computations (Sheet 2 of 2)

Operations	X	N	Z	V	C	Special Definition
ASR, LSR, ROXR	*	*	*	0	?	C = Dr – 1
ASR, LSR (r = 0)	—	*	*	0	0	
ROXR (r = 0)	—	*	*	0	?	C = X
ROR	—	*	*	0	?	C = Dr – 1
ROR (r = 0)	—	*	*	0	0	

— = Not Affected
U = Undefined, Result Meaningless
? = Other — See Special Definition
* = General Case
X = C
N = Rm
Z = $\overline{Rm}$ \ . . . \ $\overline{R0}$
Sm = Source Operand — Most Significant Bit
Dm = Destination Operand — Most Significant Bit

Rm = Result Operand — Most Significant Bit
R = Register Tested
n = Bit Number
r = Shift Count
LB = Lower bound
UB = Upper Bound
\ = Boolean AND
V = Boolean OR
$\overline{Rm}$ = NOT Rm



INSTRUCTION NAME

OPERATION DESCRIPTION

ASSEMBLER SYNTAX FOR THIS INSTRUCTION

TEXT DESCRIPTION OF INSTRUCTION OPERATION

CONDITION CODE EFFECTS (SEE 3.8 CONDITION CODES)

INSTRUCTION FORMAT — SPECIFIES THE BIT PATTERN AND FIELDS OF THE OPERATION WORD AND ANY OTHER WORDS WHICH ARE PART OF THE INSTRUCTION. THE EFFECTIVE ADDRESS EXTENSIONS ARE NOT EXPLICITLY ILLUSTRATED. THE EXTENSIONS (IF THERE ARE ANY) WOULD FOLLOW THE ILLUSTRATED PORTIONS OF THE INSTRUCTIONS. FOR THE MOVE INSTRUCTION, THE SOURCE EFFECTIVE ADDRESS EXTENSION IS THE FIRST, FOLLOWED BY THE DESTINATION EFFECTIVE ADDRESS EXTENSION.

MEANINGS AND ALLOWED VALUES OF THE VARIOUS FIELDS REQUIRED BY THE INSTRUCTION FORMAT.

ABCD

Add

Operation: Source$_{10}$ + Destination

Assembler Syntax:
ABCD Dy,Dx
ABCD −(Ay), −(Ax)

Attributes: Size = (Byte)

Description: Adds the source opera
and stores the result in the destinat
decimal arithmetic. The operands,
different ways:

1. Data register to data register:
in the instruction.
2. Memory to memory: The op
mode using the address

This operation is a byte operation

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

X Set the same as the carry bit.
N Undefined.
Z Cleared if the result is non-zero
V Undefined.
C Set if a decimal carry was gene

Normally the Z condition cod
operation. This allows success
precision operations.

Instruction Format:

15	14	13	12	11	10
1	1	0	0	REGISTER Ry	

R/M Field: 0 = Data Register to Data Register
If R/M = 0, Rx and Ry are Data Registers
If R/M = 1, Rx and Ry are Address Registers f

Instruction Fields:

Register Ry field — Specifies the d
If R/M = 0, specifies a data regi
If R/M = 1, specifies an address

R/M field — Specifies the operand
0 — the operation is data regist
1 — the operation is memory to

Figure 3-2. Instruction Description Format

ABCD Add Decimal with Extend ABCD

Operation: Source$_{10}$ + Destination$_{10}$ + X → Destination

Assembler Syntax:
ABCD Dy,Dx
ABCD −(Ay),−(Ax)

Attributes: Size = (Byte)

Description: Adds the source operand to the destination operand along with the extend bit, and stores the result in the destination location. The addition is performed using binary coded decimal arithmetic. The operands, which are packed BCD numbers, can be addressed in two different ways:

1. Data register to data register: The operands are contained in the data registers specified in the instruction.
2. Memory to memory: The operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

This operation is a byte operation only.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

X Set the same as the carry bit.
N Undefined.
Z Cleared if the result is nonzero. Unchanged otherwise.
V Undefined.
C Set if a decimal carry was generated. Cleared otherwise.

NOTE

Normally the Z condition code bit is set via programming before the start of an operation. This allows successful tests for zero results upon completion of multiple-precision operations.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER Rx			1	0	0	0	0	R/M	REGISTER Ry		

R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, Rx and Ry are Data Registers
If R/M = 1, Rx and Ry are Address Registers for the Predecrement Addressing Mode

Instruction Fields:

Register Rx field — Specifies the destination register:
If R/M = 0, specifies a data register
If R/M = 1, specifies an address register for the predecrement addressing mode

R/M field — Specifies the operand addressing mode:
0 — the operation is data register to data register
1 — the operation is memory to memory

Register Ry field — Specifies the source register:
If R/M = 0, specifies a data register
If R/M = 1, specifies an address register for the predecrement addressing mode

ADD Add ADD

Operation: Source + Destination ➧ Destination

Assembler Syntax:
ADD ⟨ea⟩,Dn
ADD Dn, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Adds the source operand to the destination operand using binary addition, and stores the result in the destination location. The size of the operation may be specified as byte, word, or long. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set the same as the carry bit.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a carry is generated. Cleared otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Register field — Specifies any of the eight data registers.

Opmode field:

Byte	Word	Long	Operation
000	001	010	⟨ea⟩ + ⟨Dn⟩ ➧ ⟨n⟩
100	101	110	⟨Dn⟩ + ⟨ea⟩ ➧ ⟨ea⟩

Effective Address Field — Determines addressing mode:

a. If the location specified is a source operand, all addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Word and Long Only

b. If the location specified is a destination operand, only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

NOTES:

1. The Dn mode is used when the destination is a data register; the destination ⟨ea⟩ mode is invalid for a data register.
2. ADDA is used when the destination is an address register. ADDI and ADDQ are used when the source is immediate data. Most assemblers automatically make this distinction.

ADDA Add Address ADDA

Operation: Source + Destination → Destination

Assembler Syntax: ADDA ⟨ea⟩, An

Attributes: Size = (Word, Long)

Description: Adds the source operand to the destination address register, and stores the result in the address register. The size of the operation may be specified as word or long. The entire destination address register is used regardless of the operation size.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Register field — Specifies any of the eight address registers. This is always the destination.

Opmode field — Specifies the size of the operation:

011 — Word operation. The source operand is sign-extended to a long operand and the operation is performed on the address register using all 32 bits.

111 — Long operation.

Effective Address field — Specifies the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

ADDI Add Immediate ADDI

Operation: Immediate Data + Destination ▶ Destination

Assembler Syntax: ADDI #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Adds the immediate data to the destination operand, and stores the result in the destination location. The size of the operation may be specified as byte, word, or long. The size of the immediate data matches the operation size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set the same as the carry bit.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a carry is generated. Cleared otherwise.

Instruction Format:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2" colspan="2">SIZE</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td colspan="8">WORD DATA (16 BITS)</td><td colspan="8">BYTE DATA (8 BITS)</td></tr>
<tr><td colspan="16">LONG DATA (32 BITS)</td></tr>
</table>

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand.
Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
− (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Immediate field — (Data immediately following the instruction):
If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is the next two immediate words.

ADDQ Add Quick ADDQ

Operation: Immediate Data + Destination → Destination

Assembler Syntax: ADDQ #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Adds an immediate value of one to eight to the operand at the destination location. The size of the operation may be specified as byte, word, or long. Word and long operations are also allowed on the address registers. When adding to address registers, the condition codes are not altered, and the entire destination address register is used regardless of the operation size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set the same as the carry bit.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Set if a carry occurs. Cleared otherwise.

The condition codes are not affected when the destination is an address register.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	DATA			0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Data field — Three bits of immediate data, 7–0 (with the immediate value 0 representing a value of 8).

Size field — Specifies the size of the operation:
00 — Byte operation
01 — Word operation
10 — Long operation

Effective Address field — Specifies the destination location.
Only alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Word and Long Only

ADDX Add Extended ADDX

Operation: Source + Destination + X → Destination

Assembler Syntax:
ADDX Dy,Dx
ADDX −(Ay),−(Ax)

Attributes: Size = (Byte, Word, Long)

Description: Adds the source operand to the destination operand along with the extend bit and stores the result in the destination location. The operands can be addressed in two different ways:

1. Data register to data register: The data registers specified in the instruction contain the operands.
2. Memory to memory: The address registers specified in the instruction address the operands using the predecrement addressing mode.

The size of the operation can be specified as byte, word, or long.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set the same as the carry bit.
N Set if the result is negative. Cleared otherwise.
Z Cleared if the result is nonzero. Unchanged otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Set if a carry is generated. Cleared otherwise.

NOTE

Normally the Z condition code bit is set via programming before the start of an operation. This allows successful tests for zero results upon completion of multiple-precision operations.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	REGISTER Rx			1	SIZE		0	0	R/M	REGISTER Ry		

Instruction Fields:

Register Rx field — Specifies the destination register:
If R/M = 0, specifies a data register.
If R/M = 1, specifies an address register for the predecrement addressing mode.

Size field — Specifies the size of the operation:
  00 — Byte operation
  01 — Word operation
  10 — Long operation
R/M field — Specifies the operand address mode:
  0 — The operation is data register to data register.
  1 — The operation is memory to memory.
Register Ry field — Specifies the source register:
  If R/M = 0, specifies a data register.
  If R/M = 1, specifies an address register for the predecrement addressing mode.

AND — AND Logical — AND

Operation: Source∧Destination ➧ Destination

Assembler Syntax:
AND ⟨ea⟩,Dn
AND Dn,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Performs an AND operation of the source operand with the destination operand and stores the result in the destination location. The size of the operation can be specified as byte, word, or long. The contents of an address register may not be used as an operand.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Register field — Specifies any of the eight data registers.

Opmode field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩)∧(⟨Dn⟩) ➧ Dn
100	101	110	(⟨Dn⟩)∧(⟨ea⟩) ➧ ea

Effective Address field — Determines addressing mode:

If the location specified is a source operand only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	111	100
(An)+	011	reg. number:An			
−(An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

If the location specified is a destination operand only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	—	—
(An)+	011	reg. number:An			
−(An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	—	—
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	—	—
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	—	—
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	—	—
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	—	—

NOTES:

1. The Dn mode is used when the destination is a data register; the destination ⟨ea⟩ mode is invalid for a data register.
2. Most assemblers use ANDI when the source is immediate data.

ANDI AND Immediate ANDI

Operation: Immediate Data $\wedge$ Destination → Destination

Assembler Syntax: ANDI #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Performs an AND operation of the immediate data with the destination operand and stores the result in the destination location. The size of the operation can be specified as byte, word, or long. The size of the immediate data matches the operation size.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">0</td><td colspan="2" rowspan="2">SIZE</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td colspan="8">WORD DATA (16 BITS)</td><td colspan="8">BYTE DATA (8 BITS)</td></tr>
<tr><td colspan="16">LONG DATA (32 BITS)</td></tr>
</table>

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand.
Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
− (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Immediate field — (Data immediately following the instruction):
If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is the next two immediate words.

ANDI to CCR — AND Immediate to Condition Codes — ANDI to CCR

Operation: Source $\wedge$ CCR $\rightarrow$ CCR

Assembler Syntax: ANDI #⟨data⟩,CCR

Attributes: Size = (Byte)

Description: Performs an AND operation of the immediate operand with the condition codes and stores the result in the low-order byte of the status register.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Cleared if bit 4 of immediate operand is zero. Unchanged otherwise.
N Cleared if bit 3 of immediate operand is zero. Unchanged otherwise.
Z Cleared if bit 2 of immediate operand is zero. Unchanged otherwise.
V Cleared if bit 1 of immediate operand is zero. Unchanged otherwise.
C Cleared if bit 0 of immediate operand is zero. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

ANDI to SR

AND Immediate to the Status Register
(Privileged Instruction)

Operation: If supervisor state
then Source $\wedge$ SR → SR
else TRAP

Assembler Syntax: ANDI #⟨data⟩,SR

Attributes: Size = (Word)

Description: Performs an AND operation of the immediate operand with the contents of the status register and stores the result in the status register. All implemented bits of the status register are affected.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Cleared if bit 4 of immediate operand is zero. Unchanged otherwise.
N Cleared if bit 3 of immediate operand is zero. Unchanged otherwise.
Z Cleared if bit 2 of immediate operand is zero. Unchanged otherwise.
V Cleared if bit 1 of immediate operand is zero. Unchanged otherwise.
C Cleared if bit 0 of immediate operand is zero. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0
WORD DATA (16 BITS)															

ASL, ASR Arithmetic Shift ASL, ASR

Operation: Destination Shifted by ⟨count⟩ → Destination

Assembler Syntax:
ASd Dx,Dy
ASd #⟨data⟩,Dy
ASd ⟨ea⟩
where d is direction, L or R

Attributes: Size = (Byte, Word, Long)

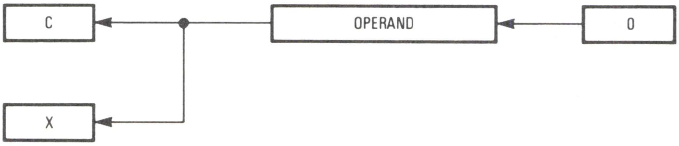
Description: Arithmetically shifts the bits of the operand in the direction (L or R) specified. The carry bit receives the last bit shifted out of the operand. The shift count for the shifting of a register may be specified in two different ways:

1. Immediate — The shift count is specified in the instruction (shift range, 8–1).
2. Register — The shift count is the value in the data register specified in instruction modulo 64.

The size of the operation can be specified as byte, word, or long. An operand in memory can be shifted one bit only, and the operand size is restricted to a word.

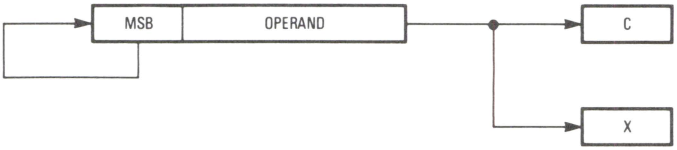
For ASL, the operand is shifted left; the number of positions shifted is the shift count. Bits shifted out of the high-order bit go to both the carry and the extend bits; zeros are shifted into the low-order bit. The overflow bit indicates if any sign changes occur during the shift.

ASL:



For ASR, the operand is shifted right; the number of positions shifted is the shift count. Bits shifted out of the low-order bit go to both the carry and the extend bits; the sign-bit (MSB) is shifted into the high-order bit.

ASR:



Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set according to the last bit shifted out of the operand. Unaffected for a shift count of zero.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if the most significant bit is changed at any time during the shift operation. Cleared otherwise.
C Set according to the last bit shifted out of the operand. Cleared for a shift count of zero.

Instruction Format (Register Shifts):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	COUNT/REGISTER			dr	SIZE		i/r	0	0	REGISTER		

Instruction Fields (Register Shifts):

Count/Register field — Specifies shift count or register that contains the shift count:
  If i/r = 0, this field contains the shift count. The values one to seven represent counts of one to seven; value of zero represents a count of eight.
  If i/r = 1, this field specifies the data register that contains the shift count (modulo 64).

dr field — Specifies the direction of the shift:
  0 — Shift right
  1 — Shift left

Size field — Specifies the size of the operation:
  00 — Byte operation
  01 — Word operation
  10 — Long operation

i/r field:
  If i/r = 0, specifies immediate shift count.
  If i/r = 1, specifies register shift count.

Register field — Specifies a data register to be shifted.

ASL, ASR Arithmetic Shift ASL, ASR

Instruction Format (Memory Shifts):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Memory Shifts):

dr field — Specifies the direction of the shift:

0 — Shift right

1 — Shift left

Effective Address field — Specifies the operand to be shifted.

Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Bcc Branch Conditionally Bcc

Operation: If (condition true) then PC + d ➧ PC

Assembler Syntax: Bcc ⟨label⟩

Attributes: Size = (Byte, Word, Long)

Description: If the specified condition is true, program execution continues at location (PC) + displacement. The PC contains the address of the instruction word of the Bcc instruction plus two. The displacement is a twos complement integer that represents the relative distance in bytes from the current PC to the destination PC. If the 8-bit displacement field in the instruction word is zero, a 16-bit displacement (the word immediately following the instruction) is used. If the 8-bit displacement field in the instruction word is all ones ($FF), the 32-bit displacement (long word immediately following the instruction) is used. Condition code cc specifies one of the following conditions:

CC	carry clear	0100	$\overline{C}$	LS	low or same	0011	$C+Z$
CS	carry set	0101	C	LT	less than	1101	$N\cdot\overline{V}+\overline{N}\cdot V$
EQ	equal	0111	Z	MI	minus	1011	N
GE	greater or equal	1100	$N\cdot V+\overline{N}\cdot\overline{V}$	NE	not equal	0110	$\overline{Z}$
GT	greater than	1110	$N\cdot V\cdot\overline{Z}+\overline{N}\cdot\overline{V}\cdot\overline{Z}$	PL	plus	1010	$\overline{N}$
HI	high	0010	$\overline{C}\cdot\overline{Z}$	VC	overflow clear	1000	$\overline{V}$
LE	less or equal	1111	$Z+N\cdot\overline{V}+\overline{N}\cdot V$	VS	overflow set	1001	V

Condition Codes:

Not affected.

Instruction Format:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td colspan="4">CONDITION</td><td colspan="8">8-BIT DISPLACEMENT</td></tr>
<tr><td colspan="16">16-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $00</td></tr>
<tr><td colspan="16">32-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $FF</td></tr>
</table>

Instruction Fields:

Condition field — The binary code for one of the conditions listed in the table.

8-Bit Displacement field — Twos complement integer specifying the number of bytes between the branch instruction and the next instruction to be executed if the condition is met.

16-Bit Displacement field — Used for the displacement when the 8-bit displacement field contains $00.

32-Bit Displacement field — Used for the displacement when the 8-bit displacement field contains $FF.

NOTE

A branch to the immediately following instruction automatically uses the 16-bit displacement format because the 8-bit displacement field contains $00 (zero offset).

BCHG — Test a Bit and Change — BCHG

Operation: ~(⟨number⟩ of Destination) → Z;
~(⟨number⟩ of Destination) → ⟨bit number⟩ of Destination

Assembler Syntax: BCHG Dn,⟨ea⟩
BCHG #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Long)

Description: Tests a bit in the destination operand and sets the Z condition code appropriately, then inverts the specified bit in the destination. When the destination is a data register, any of the 32 bits can be specified by the modulo 32-bit number. When the destination is a memory location, the operation is a byte operation, and the bit number is modulo 8. In all cases, bit zero refers to the least significant bit. The bit number for this operation may be specified in either of two ways:

1. Immediate — The bit number is specified in a second word of the instruction.
2. Register — The specified data register contains the bit number.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

X Not affected.
N Not affected.
Z Set if the bit tested is zero. Cleared otherwise.
V Not affected.
C Not affected.

Instruction Format (Bit Number Dynamic, specified in a register):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	REGISTER			1	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Bit Number Dynamic):

Register field — Specifies the data register that contains the bit number.

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
− (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

*Long only; all others are byte only.

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Instruction Format (Bit Number Static, specified as immediate data):

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="8">BIT NUMBER</td></tr>
</table>

Instruction Fields (Bit Number Static):

Effective Address field — Specifies the destination location.

Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
− (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

*Long only; all others are byte only.

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Bit Number field — Specifies the bit number.

BCLR Test a Bit and Clear BCLR

Operation: ~(⟨bit number⟩ of Destination) → Z;
0 → ⟨bit number⟩ of Destination

Assembler Syntax:
BCLR Dn,⟨ea⟩
BCLR #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Long)

Description: Tests a bit in the destination operand and sets the Z condition code appropriately, then clears the specified bit in the destination. When a data register is the destination, any of the 32 bits can be specified by a modulo 32-bit number. When a memory location is the destination, the operation is a byte operation, and the bit number is modulo 8. In all cases, bit zero refers to the least significant bit. The bit number for this operation can be specified in either of two ways:

1. Immediate — The bit number is specified in a second word of the instruction.
2. Register — The specified data register contains the bit number.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

X Not affected.
N Not affected.
Z Set if the bit tested is zero. Cleared otherwise.
V Not affected.
C Not affected.

Instruction Format (Bit Number Dynamic, specified in a register):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	REGISTER			1	1	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Bit Number Dynamic):

Register field — Specifies the data register that contains the bit number.

Effective Address field — Specifies the destination location.

Only data alterable addressing modes are allowed as shown:

BCLR Test a Bit and Clear BCLR

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Long only; all others are byte only.

Instruction Format (Bit Number Static, specified as immediate data):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	0	0	0	0	0	0	0	BIT NUMBER							

Instruction Fields (Bit Number Static):

Effective Address field — Specifies the destination location.

Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Long only; all others are byte only.

Bit Number field — Specifies the bit number.

BFCHG Test Bit Field and Change BFCHG

Operation: ~(⟨bit field⟩ of Destination) ▶ ⟨bit field⟩ of Destination

Assembler Syntax: BFCHG ⟨ea⟩{offset:width}

Attributes: Unsized

Description: Sets the condition codes according to the value in a bit field at the specified effective address, then complements the field.

A field offset and a field width select the field. The field offset specifies the starting bit of the field. The field width determines the number of bits in the field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the field is set. Cleared otherwise.
Z Set if all bits of the field are zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	1	1	EFFECTIVE ADDRESS MODE			REGISTER		
0	0	0	0	Do	OFFSET					Dw	WIDTH				

Instruction Fields:

Effective Address field — Specifies the base location for the bit field. Only data register direct or control alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	—	—
− (An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Do field — Determines how the field offset is specified.

0 — The Offset field contains the bit field offset.

1 — Bits [8:6] of the extension word specify a data register that contains the offset; bits [10:9] are 0.

Offset field — Specifies the field offset, depending on Do.

If Do = 0, the Offset field is an immediate operand; the operand value is in the range 0-31.

If Do = 1, the Offset field specifies a data register that contains the offset. The value is in the range of -2^{31} to $2^{31}-1$.

Dw field — Determines how the field width is specified.

0 — The Width field contains the bit field width.

1 — Bits [2:0] of the extension word specify a data register that contains the width; bits [3:4] are 0.

Width field — Specifies the field width, depending on Dw.

If Dw = 0, the Width field is an immediate operand; an operand value in the range 1–31 specifies a field width of 1–31, and a value of 0 specifies a width of 32.

If Dw = 1, the Width field specifies a data register that contains the width. The value is modulo 32; values of 1–31 specify field widths of 1–31, and a value of 0 specifies a width of 32.

BFCLR Test Bit Field and Clear BFCLR

Operation: 0 → ⟨bit field⟩ of Destination

Assembler Syntax: BFCLR ⟨ea⟩{offset:width}

Attributes: Unsized

Description: Sets condition codes according to the value in a bit field at the specified effective address, and clears the field.

The field offset and field width select the field. The field offset specifies the starting bit of the field. The field width determines the number of bits in the field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the field is set. Cleared otherwise.
Z Set if all bits of the field are zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>Do</td><td colspan="5">OFFSET</td><td>Dw</td><td colspan="5">WIDTH</td></tr>
</table>

Instruction Fields:

Effective Address field — Specifies the base location for the bit field. Only data register direct or alterable control addressing modes are allowed, as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	—	—
−(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Do field — Determines how the field offset is specified.

0 — The Offset field contains the bit field offset.

1 — Bits [8:6] of the extension word specify a data register that contains the offset; bits [10:9] are 0.

Offset field — Specifies the field offset, depending on Do.

If Do = 0, the Offset field is an immediate operand; the operand value is in the range of 0-31.

If Do = 1, the Offset field specifies a data register that contains the offset. The value is in the range of -2^{31} to $2^{31}-1$.

Dw field — Determines how the field width is specified.

0 — The Width field contains the bit field width.

1 — Bits [2:0] of the extension word specify a data register that contains the width; bits [3:4] are 0.

Width field — Specifies the field width, depending on Dw.

If Dw = 0, the Width field is an immediate operand; operand values in the range of 1–31 specify a field width of 1–31, and a value of 0 specifies a width of 32.

If Dw = 1, the Width field specifies a data register that contains the width. The value is modulo 32; values of 1–31 specify field widths of 1–31, and a value of 0 specifies a width of 32.

BFEXTS Extract Bit Field Signed BFEXTS

Operation: ⟨bit field⟩ of Source → Dn

Assembler Syntax: BFEXTS ⟨ea⟩{offset:width},Dn

Attributes: Unsized

Description: Extracts a bit field from the specified effective address location, sign extends to 32 bits, and loads the result into the destination data register.

The field offset and field width select the bit field. The field offset specifies the starting bit of the field. The field width determines the number of bits in the field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the field is set. Cleared otherwise.
Z Set if all bits of the field are zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER			Do	OFFSET					Dw	WIDTH				

Instruction Fields:

Effective Address field — Specifies the base location for the bit field. Only data register direct or control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	—	—
−(An)	—	—
(d$_{16}$,An)	101	reg. number:An
(d$_{8}$,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d$_{16}$,PC)	111	010
(d$_{8}$,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Register field — Specifies the destination register.

Do field — Determines how the field offset is specified.

0 — The Offset field contains the bit field offset.

1 — Bits [8:6] of the extension word specify a data register that contains the offset; bits [10:9] are 0.

Offset field — Specifies the field offset, depending on Do.

If Do = 0, the Offset field is an immediate operand; the operand value is in the range of 0-31.

If Do = 1, the Offset field specifies a data register that contains the offset. The value is in the range of -2^{31} to $2^{31}-1$.

Dw field — Determines how the field width is specified.

0 — The Width field contains the bit field width.

1 — Bits [2:0] of the extension word specify a data register that contains the width; bits [4:3] are 0.

Width field — Specifies the field width, depending on Dw.

If Dw = 0, the Width field is an immediate operand; operand values in the range of 1–31 specify a field width of 1–31, and a value of 0 specifies a width of 32.

If Dw = 1, the Width field specifies a data register that contains the width. The value is modulo 32; values of 1–31 specify field widths of 1–31, and a value of 0 specifies a width of 32.

BFEXTU Extract Bit Field Unsigned BFEXTU

Operation: ⟨bit offset⟩ of Source → Dn

Assembler Syntax: BFEXTU ⟨ea⟩{offset:width},Dn

Attributes: Unsized

Description: Extracts a bit field from the specified effective address location, zero extends to 32 bits, and loads the results into the destination data register.

The field offset and field width select the field. The field offset specifies the starting bit of the field. The field width determines the number of bits in the field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the source field is set. Cleared otherwise.
Z Set if all bits of the field are zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER			Do	OFFSET					Dw	WIDTH				

BFEXTU Extract Bit Field Unsigned BFEXTU

Instruction Fields:

Effective Address field — Specifies the base location for the bit field. Only data register direct or control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	—	—
−(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Register field — Specifies the destination data register.

Do field — Determines how the field offset is specified.

0 — The Offset field contains the bit field offset.

1 — Bits [8:6] of the extension word specify a data register that contains the offset; bits [10:9] are 0.

Offset field — Specifies the field offset, depending on Do.

If Do = 0, the Offset field is an immediate operand; the operand value is in the range of 0-31.

If Do = 1, the Offset field specifies a data register that contains the offset. The value is in the range of -2^{31} to $2^{31}-1$.

Dw field — Determines how the field width is specified.

0 — The Width field contains the bit field width.

1 — Bits [2:0] of the extension word specify a data register that contains the width; bits [4:3] are 0.

Width field — Specifies the field width, depending on Dw.

If Dw = 0, the Width field is an immediate operand; operand values in the range of 1–31 specify a field width of 1–31, and a value of 0 specifies a width of 32.

If Dw = 1, the Width field specifies a data register that contains the width. The value is modulo 32; values of 1–31 specify field widths of 1–31, and a value of 0 specifies a width of 32.

BFFFO Find First One in Bit Field BFFFO

Operation: ⟨bit offset⟩ of Source Bit Scan → Dn

Assembler Syntax: BFFFO ⟨ea⟩{offset:width},Dn

Attributes: Unsized

Description: Searches the source operand for the most significant bit that is set to a value of one. The bit offset of that bit (the bit offset in the instruction plus the offset of the first one bit) is placed in Dn. If no bit in the bit field is set to one, the value in Dn is the field offset plus the field width. The instruction sets the condition codes according to the bit field value.

The field offset and field width select the field. The field offset specifies the starting bit of the field. The field width determines the number of bits in the field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the field is set. Cleared otherwise.
Z Set if all bits of the field are zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER			Do	OFFSET					Dw	WIDTH				

Instruction Fields:

Effective Address field — Specifies the base location for the bit field. Only data register direct or control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	—	—
−(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Register field — Specifies the destination data register operand.

Do field — Determines how the field offset is specified.

0 — The Offset field contains the bit field offset.

1 — Bits [8:6] of the extension word specify a data register that contains the offset; bits [10:9] are 0.

Offset field — Specifies the field offset, depending on Do.

If Do = 0, the Offset field is an immediate operand; the operand value is in the range of 0–31.

If Do = 1, the Offset field specifies a data register that contains the offset. The value is in the range of -2^{31} to $2^{31}-1$.

Dw field — Determines how the field width is specified.

0 — The Width field contains the bit field width.

1 — Bits [2:0] of the extension word specify a data register that contains the width; bits [4:3] are 0.

Width field — Specifies the field width, depending on Dw.

If Dw = 0, the Width field is an immediate operand; operand values in the range of 1–31 specify a field width of 1–31, and a value of 0 specifies a width of 32.

If Dw = 1, the Width field specifies a data register that contains the width. The value is modulo 32; values of 1–31 specify field widths of 1–31, and a value of 0 specifies a width of 32.

BFINS Insert Bit Field BFINS

Operation: Dn → ⟨bit field⟩ of Destination

Assembler Syntax: BFINS Dn,⟨ea⟩{offset:width}

Attributes: Unsized

Description: Inserts a bit field taken from the low-order bits of the specified data register into a bit field at the effective address location. The instruction sets the condition codes according to the inserted value.

The field offset and field width select the field. The field offset specifies the starting bit of the field. The field width determines the number of bits in the field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the field is set. Cleared otherwise.
Z Set if all bits of the field are zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td>0</td><td colspan="3">REGISTER</td><td>Do</td><td colspan="5">OFFSET</td><td>Dw</td><td colspan="5">WIDTH</td></tr>
</table>

Instruction Fields:

Effective Address field — Specifies the base location for the bit field. Only data register direct or control alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	—	—
−(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Register field — Specifies the source data register operand.

Do field — Determines how the field offset is specified.
- 0 — The Offset field contains the bit field offset.
- 1 — Bits [8:6] of the extension word specify a data register that contains the offset; bits [10:9] are 0.

Offset field — Specifies the field offset, depending on Do.
- If Do = 0, the Offset field is an immediate operand; the operand value is in the range of 0-31.
- If Do = 1, the Offset field specifies a data register that contains the offset. The value is in the range of -2^{31} to $2^{31}-1$.

Dw field — Determines how the field width is specified.
- 0 — The Width field contains the bit field width.
- 1 — Bits [2:0] of the extension word specify a data register that contains the width; bits [4:3] are 0.

Width field — Specifies the field width, depending on Dw.
- If Dw = 0, the Width field is an immediate operand; operand values in the range of 1–31 specify a field width of 1–31, and a value of 0 specifies a width of 32.
- If Dw = 1, the Width field specifies a data register that contains the width. The value is modulo 32; values of 1–31 specify field widths of 1–31, and a value of 0 specifies a width of 32.

BFSET Test Bit Field and Set BFSET

Operation: 1s → ⟨bit field⟩ of Destination

Assembler Syntax: BFSET ⟨ea⟩{offset:width}

Attributes: Unsized

Description: Sets the condition codes according to the value in a bit field at the specified effective address, then sets each bit in the field.

The field offset and the field width select the field. The field offset specifies the starting bit of the field. The field width determines the number of bits in the field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the field is set. Cleared otherwise.
Z Set if all bits of the field are zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	EFFECTIVE ADDRESS MODE			REGISTER		
0	0	0	0	Do	OFFSET					Dw	WIDTH				

Instruction Fields:

Effective Address field — Specifies the base location for the bit field. Only data register direct or control alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	—	—
−(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Do field — Determines how the field offset is specified.
0 — The Offset field contains the bit field offset.
1 — Bits [8:6] of the extension word specify a data register that contains the offset; bits [10:9] are 0.

Offset field — Specifies the field offset, depending on Do.
If Do = 0, the Offset field is an immediate operand; the operand value is in the range of 0-31.
If Do = 1, the Offset field specifies a data register that contains the offset. The value is in the range of -2^{31} to $2^{31}-1$.

Dw field — Determines how the field width is specified.
0 — The Width field contains the bit field width.
1 — Bits [2:0] of the extension word specify a data register that contains the width; bits [4:3] are 0.

Width field — Specifies the field width, depending on Dw.
If Dw = 0, the Width field is an immediate operand; operand values in the range of 1–31 specify a field width of 1–31, and a value of 0 specifies a width of 32.
If Dw = 1, the Width field specifies a data register that contains the width. The value is modulo 32; values of 1–31 specify field widths of 1–31, and a value of 0 specifies a width of 32.

BFTST Test Bit Field BFTST

Operation: ⟨bit field⟩ of Destination

Assembler Syntax: BFTST ⟨ea⟩{offset:width}

Attributes: Unsized

Description: Sets the condition codes according to the value in a bit field at the specified effective address location.

The field offset and field width select the field. The field offset specifies the starting bit of the field. The field width determines the number of bits in the field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the field is set. Cleared otherwise.
Z Set if all bits of the field are zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	0	0	0	Do	OFFSET					Dw	WIDTH				

Instruction Fields:

Effective Address field — Specifies the base location for the bit field. Only data register direct or control addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	—	—
(An)+	—	—			
−(An)	—	—			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

Do field — Determines how the field offset is specified.

0 — The Offset field contains the bit field offset.

1 — Bits [8:6] of the extension word specify a data register that contains the offset; bits [10:9] are 0.

Offset field — Specifies the field offset, depending on Do.

If Do = 0, the Offset field is an immediate operand; the operand value is in the range of 0–31.

If Do = 1, the Offset field specifies a data register that contains the offset. The value is in the range of -2^{31} to $2^{31}-1$.

Dw field — Determines how the field width is specified.

0 — The Width field contains the bit field width.

1 — Bits [2:0] of the extension word specify a data register that contains the width; bits [4:3] are 0.

Width field — Specifies the field width, depending on Dw.

If Dw = 0, the Width field is an immediate operand, operand values in the range of 1–31 specify a field width of 1–31, and a value of 0 specifies a width of 32.

If Dw = 1, the Width field specifies a data register that contains the width. The value is modulo 32; values of 1–31 specify field widths of 1–31, and a value of 0 specifies a width of 32.

BKPT Breakpoint BKPT

Operation:

```
Run breakpoint acknowledge cycle
If acknowledged
   then execute returned operation word
   else TRAP as illegal instruction
```

Assembler Syntax: BKPT #⟨data⟩

Attributes: Unsized

Description: Executes a breakpoint acknowledge bus cycle with the immediate data (value 0–7) on bits 2–4 of the address bus and zeros on bits 0 and 1 of the address bus.

The breakpoint acknowledge cycle accesses the CPU space, addressing type 0, and provides the breakpoint number specified by the instruction on address lines A4–A2. If the external hardware terminates the cycle with $\overline{\text{DSACKxs}}$ or $\overline{\text{STERM}}$, the data on the bus (an instruction word) is inserted into the instruction pipe and is executed after the breakpoint instruction. The breakpoint instruction requires a word to be transferred so if the first bus cycle accesses an 8-bit port, a second cycle is required. If the external logic terminates the breakpoint acknowledge cycle with $\overline{\text{BERR}}$ (i.e., no instruction word available) the processor takes an illegal instruction exception. Refer to **7.4.2 Breakpoint Acknowledge** for details of breakpoint acknowledge cycle operation.

This instruction supports breakpoints for debug monitors and real-time hardware emulators. The exact operation performed by the instruction is implementation-dependent. Typically, this instruction replaces an instruction in a program; that instruction is returned by the breakpoint acknowledge cycle.

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	1	VECTOR		

Instruction Fields:

Vector field — Contains the immediate data, a value in the range of 0–7. This is the breakpoint number.

BRA Branch Always BRA

Operation: PC + d ➧ PC

Assembler Syntax: BRA ⟨label⟩

Attributes: Size = (Byte, Word, Long)

Description: Program execution continues at location (PC) + displacement. The PC contains the address of the instruction word of the BRA instruction plus two. The displacement is a twos complement integer that represents the relative distance in bytes from the current PC to the destination PC. If the 8-bit displacement field in the instruction word is zero, a 16-bit displacement (the word immediately following the instruction) is used. If the 8-bit displacement field in the instruction word is all ones ($FF), the 32-bit displacement (long word immediately following the instruction) is used.

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	8-BIT DISPLACEMENT							
16-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $00															
32-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $FF															

Instruction Fields:

8-Bit Displacement field — Twos complement integer specifying the number of bytes between the branch instruction and the next instruction to be executed.

16-Bit Displacement field — Used for a larger displacement when the 8-bit displacement is equal to $00.

32-Bit Displacement field — Used for a larger displacement when the 8-bit displacement is equal to $FF.

NOTE

A branch to the immediately following instruction automatically uses the 16-bit displacement format because the 8-bit displacement field contains $00 (zero offset).

BSET Test a Bit and Set BSET

Operation: ~(⟨bit number⟩ of Destination) → Z;
1 → ⟨bit number⟩ of Destination

Assembler Syntax:
BSET Dn,⟨ea⟩
BSET #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Long)

Description: Tests a bit in the destination operand and sets the Z condition code appropriately. Then sets the specified bit in the destination operand. When a data register is the destination, any of the 32 bits can be specified by a modulo 32-bit number. When a memory location is the destination, the operation is a byte operation, and the bit number is modulo 8. In all cases, bit zero refers to the least significant bit. The bit number for this operation can be specified in either of two ways:

1. Immediate — The bit number is specified in the second word of the instruction.
2. Register — The specified data register contains the bit number.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

X Not affected.
N Not affected.
Z Set if the bit tested is zero. Cleared otherwise.
V Not affected.
C Not affected.

Instruction Format (Bit Number Dynamic, specified in a register):

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="3" rowspan="2">REGISTER</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Instruction Fields (Bit Number Dynamic):

Register field — Specifies the data register that contains the bit number.

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d$_{16}$,An)	101	reg. number:An
(d$_{8}$,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d$_{16}$,PC)	—	—
(d$_{8}$,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Long only; all others are byte only.

Instruction Format (Bit Number Static, specified as immediate data):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	0	0	0	0	0	0	BIT NUMBER								

Instruction Fields (Bit Number Static):

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d$_{16}$,An)	101	reg. number:An
(d$_{8}$,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d$_{16}$,PC)	—	—
(d$_{8}$,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Long only; all others are byte only.

Bit Number field — Specifies the bit number.

BSR Branch to Subroutine BSR

Operation: SP − 4 → SP; PC → (SP); PC + d → PC

Assembler Syntax: BSR ⟨label⟩

Attributes: Size = (Byte, Word, Long)

Description: Pushes the long word address of the instruction immediately following the BSR instruction onto the system stack. The PC contains the address of the instruction word plus two. Program execution then continues at location (PC) + displacement. The displacement is a twos complement integer that represents the relative distance in bytes from the current PC to the destination PC. If the 8-bit displacement field in the instruction word is zero, a 16-bit displacement (the word immediately following the instruction) is used. If the 8-bit displacement field in the instruction word is all ones ($FF), the 32-bit displacement (long word immediately following the instruction) is used.

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1	8-BIT DISPLACEMENT							
16-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $00															
32-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $FF															

Instruction Fields:

8-Bit Displacement field — Twos complement integer specifying the number of bytes between the branch instruction and the next instruction to be executed.

16-Bit Displacement field -- Used for a larger displacement when the 8-bit displacement is equal to $00.

32-Bit Displacement field — Used for a larger displacement when the 8-bit displacement is equal to $FF.

NOTE

A branch to the immediately following instruction automatically uses the 16-bit displacement format because the 8-bit displacement field contains $00 (zero offset).

BTST Test a Bit BTST

Operation: –(⟨bit number⟩ of Destination) → Z;

Assembler Syntax:
BTST Dn,⟨ea⟩
BTST #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Long)

Description: Tests a bit in the destination operand and sets the Z condition code appropriately. When a data register is the destination, any of the 32 bits can be specified by a modulo 32 bit number. When a memory location is the destination, the operation is a byte operation, and the bit number is modulo 8. In all cases, bit zero refers to the least significant bit. The bit number for this operation can be specified in either of two ways:

1. Immediate — The bit number is specified in a second word of the instruction.
2. Register — The specified data register contains the bit number.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

X Not affected.
N Not affected.
Z Set if the bit tested is zero. Cleared otherwise.
V Not affected.
C Not affected.

Instruction Format (Bit Number Dynamic, specified in a register):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										EFFECTIVE ADDRESS					
0	0	0	0	REGISTER			1	0	0	MODE			REGISTER		

Instruction Fields (Bit Number Dynamic):

Register field — Specifies the data register that contains the bit number.

Effective Address field — Specifies the destination location. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Long only; all others are byte only.

Instruction Format (Bit Number Static, specified as immediate data):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	0	0	0	0	0	0	0	BIT NUMBER							

Instruction Fields (Bit Number Static):

Effective Address field — Specifies the destination location. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Bit Number field — Specifies the bit number.

CALLM CALL Module CALLM

Operation: Save current module state on stack;
Load new module state from destination

Assembler Syntax: CALLM #<data>, <ea>

Attributes: Unsized

Description: The effective address of the instruction is the location of an external module descriptior. A module frame is created on the top of the stack, and the current module state is saved in the frame. The immediate operand specifies the number of bytes of arguments to be passed to the called module. A new module state is loaded from the descriptor addressed by the effective address. Additional information is presented in **SECTION 11 APPLICATIONS INFORMATION**.

Condition Codes: Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										EFFECTIVE ADDRESS					
0	0	0	0	0	1	1	0	1	1	MODE			REGISTER		
0	0	0	0	0	0	0	0	ARGUMENT COUNT							

Instruction Fields:

Effective Address field — Specifies the address of the module descriptor. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
-(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Argument Count field — Specifies the number of bytes of arguments to be passed to the called module. The 8-bit field can specify from 0 to 255 bytes of arguments. The same number of bytes is removed from the stack by the RTM instruction.

CAS
CAS2

Compare and Swap with Operand

CAS
CAS2

Operation: CAS Destination — Compare Operand ➧ cc;
if Z, Update Operand ➧ Destination
else Destination ➧ Compare Operand

CAS2 Destination 1 — Compare 1 ➧ cc;
if Z, Destination 2 — Compare 2 ➧ cc
if Z, Update 1 ➧ Destination 1; Update 2 ➧ Destination 2
else Destination 1 ➧ Compare 1; Destination 2 ➧ Compare 2

Assembler Syntax: CAS Dc,Du,⟨ea⟩
CAS2 Dc1:Dc2,Du1:Du2,(Rn1):(Rn2)

Attributes: Size = (Byte*, Word, Long)

Description: CAS compares the effective address operand to the compare operand (Dc). If the operands are equal, the instruction writes the update operand (Du) to the effective address operand; otherwise, the instruction writes the effective address operand to the compare operand (Dc).

CAS2 compares memory operand 1 (Rn1) to compare operand 1 (Dc1). If the operands are equal, the instruction compares memory operand 2 (Rn2) to compare operand 2 (Dc2). If these operands are also equal, the instruction writes the update operands (Du1 and Du2) to the memory operands (Rn1 and Rn2). If either comparison fails, the instruction writes the memory operands (Rn1 and Rn2) to the compare operands (Dc1 and Dc2).

Both operations access memory using read-modify-write cycles; these instructions cannot be interrupted. This provides a means of synchronizing several processors. Typical applications of these instructions are discussed in **3.4 USING THE CAS AND CAS2 INSTRUCTIONS**.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a borrow is generated. Cleared otherwise.

---

*CAS2 cannot use byte operands

CAS CAS2 Compare and Swap with Operand CAS CAS2

Instruction Format: (CAS):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	SIZE		0	1	1	EFFECTIVE ADDRESS MODE			REGISTER		
0	0	0	0	0	0	0	Du			0	0	0	Dc		

Instruction Fields:

Size field — Specifies the size of the operation.
- 01 — Byte operation
- 10 — Word operation
- 11 — Long operation

Effective Address field — Specifies the location of the memory operand. Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	—	—
(An)+	011	reg. number:An			
−(An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	—	—
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	—	—
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	—	—
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	—	—
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	—	—

Du field — Specifies the data register that contains the update value to be written to the memory operand location if the comparison is successful.

Dc field — Specifies the data register that contains the value to be compared to the memory operand.

CAS
CAS2

Compare and Swap with Operand

CAS
CAS2

Instruction Format (CAS2):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	SIZE		0	1	1	1	1	1	1	0	0
D/A1	Rn1			0	0	0	Du1			0	0	0	Dc1		
D/A2	Rn2			0	0	0	Du2			0	0	0	Dc2		

Instruction Fields:

Size field — Specifies the size of the operation.
- 10 — Word operation.
- 11 — Long operation.

D/A1, D/A2 fields — Specify whether Rn1 and Rn2 reference data or address registers, respectively.
- 0 — The corresponding register is a data register.
- 1 — The corresponding register is an address register.

Rn1, Rn2 fields — Specify the numbers of the registers that contain the addresses of the first and second memory operands, respectively. If the operands overlap in memory, the results of any memory update are undefined.

Du1, Du2 fields — Specify the data registers that contain the update values to be written to the first and second memory operand locations if the comparison is successful.

Dc1, Dc2 fields — Specify the data registers that contain the test values to be compared to the first and second memory operands, respectively. If Dc1 and Dc2 specify the same data register and the comparison fails, memory operand 1 is stored in the data register.

Programming Note:

The CAS and CAS2 instructions can be used to perform secure update operations on system control data structures in a multiprocessing environment.

CHK Check Register Against Bounds CHK

Operation: If Dn < 0 or Dn > Source then TRAP

Assembler Syntax: CHK ⟨ea⟩,Dn

Attributes: Size = (Word, Long)

Description: Compares the value in the data register specified in the instruction to zero and to the upper bound (effective address operand). The upper bound is a twos complement integer. If the register value is less than zero or greater than the upper bound, a CHK instruction exception, vector number 6, occurs.

Condition Codes:

X	N	Z	V	C
—	*	U	U	U

X Not affected.
N Set if Dn < 0; cleared if Dn > effective address operand. Undefined otherwise.
Z Undefined.
V Undefined.
C Undefined.

Instruction Format:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="3" rowspan="2">REGISTER</td><td colspan="2" rowspan="2">SIZE</td><td rowspan="2">0</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Instruction Fields:

Register field — Specifies the data register that contains the value to be checked.
Size field — Specifies the size of the operation.
- 11 — Word operation.
- 10 — Long operation.

Effective Address field — Specifies the upper bound operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

CHK2 Check Register Against Bounds CHK2

Operation: If Rn < lower bound or
Rn > upper bound
then TRAP

Assembler Syntax: CHK2 ⟨ea⟩,Rn

Attributes: Size = (Byte, Word, Long)

Description: Compares the value in Rn to each bound. The effective address contains the bounds pair: the lower bound followed by the upper bound. For signed comparisons, the arithmetically smaller value should be used as the lower bound. For unsigned comparisons, the logically smaller value should be the lower bound.

The size of the data and the bounds can be specified as byte, word, or long. If Rn is a data register and the operation size is byte or word, only the appropriate low-order part of Rn is checked. If Rn is an address register and the operation size is byte or word, the bounds operands are sign-extended to 32 bits and the resultant operands are compared to the full 32 bits of An.

If the upper bound equals the lower bound, the valid range is a single value. If the register value is less than the lower bound or greater than the upper bound, a CHK instruction exception, vector number 6, occurs.

Condition Codes:

X	N	Z	V	C
—	U	*	U	*

X Not affected.
N Undefined.
Z Set if Rn is equal to either bound. Cleared otherwise.
V Undefined.
C Set if Rn is out of bounds. Cleared otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	SIZE		0	1	1	EFFECTIVE ADDRESS MODE			REGISTER		
D/A	REGISTER			1	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields:

Size field — Specifies the size of the operation.
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the location of the bounds operands. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
−(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

D/A field — Specifies whether an address register or data register is to be checked.
- 0 — Data register.
- 1 — Address register.

Register field — Specifies the address or data register that contains the value to be checked.

CLR Clear an Operand CLR

Operation: 0 → Destination

Assembler Syntax: CLR ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Clears the destination operand to zero. The size of the operation may be specified as byte, word, or long.

Condition Codes:

X	N	Z	V	C
—	0	1	0	0

X Not affected.
N Always cleared.
Z Always set.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Size field — Specifies the size of the operation.
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

CMP Compare CMP

Operation: Destination — Source → cc

Assembler Syntax: CMP ⟨ea⟩, Dn

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the source operand from the destination data register and sets the condition codes according to the result; the data register is not changed. The size of the operation can be byte, word, or long.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Set if a borrow occurs. Cleared otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	REGISTER			OP-MODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Register field — Specifies the destination data register.

Op-Mode field:

Byte	Word	Long	Operation
000	001	010	(⟨Dn⟩) – (⟨ea⟩)

Effective Address field — Specifies the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

*Word and Long only.

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

NOTE

CMPA is used when the destination is an address register. CMPI is used when the source is immediate data. CMPM is used for memory-to-memory compares. Most assemblers automatically make the distinction.

CMPA Compare Address CMPA

Operation: Destination − Source

Assembler Syntax: CMPA ⟨ea⟩, An

Attributes: Size = (Word, Long)

Description: Subtracts the source operand from the destination address register and sets the condition codes according to the result; the address register is not changed. The size of the operation can be specified as word or long. Word length source operands are sign extended to 32-bits for comparison.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a borrow is generated. Cleared otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Opmode Field:

Word	Long	Operation

Instruction Fields:

Register field — Specifies the destination address register.

Op-Mode field — Specifies the size of the operation:

011 — Word operation. The source operand is sign-extended to a long operand and the operation is performed on the address register using all 32 bits.

111 — Long operation.

Effective Address field — Specifies the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

CMPI Compare Immediate CMPI

Operation: Destination − Immediate Data

Assembler Syntax: CMPI #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the immediate data from the destination operand and sets the condition codes according to the result; the destination location is not changed. The size of the operation may be specified as byte, word, or long. The size of the immediate data matches the operation size.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Set if a borrow occurs. Cleared otherwise.

Instruction Format:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="2" rowspan="2">SIZE</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td colspan="8">WORD DATA (16 BITS)</td><td colspan="8">BYTE DATA (8 BITS)</td></tr>
<tr><td colspan="16">LONG DATA (32 BITS)</td></tr>
</table>

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. Only data addressing modes, except immediate are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Immediate field — (Data immediately following the instruction):
If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is the next two immediate words.

CMPM Compare Memory CMPM

Operation: Destination — Source ▶ cc

Assembler Syntax: CMPM (Ay)+,(Ax)+

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the source operand from the destination operand and sets the condition codes according to the results; the destination location is not changed. The operands are always addressed with the postincrement addressing mode, using the address registers specified in the instruction. The size of the operation may be specified as byte, word, or long.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a borrow is generated. Cleared otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	REGISTER Ax			1	SIZE		0	0	1	REGISTER Ay		

Instruction Fields:

Register Ax field — (always the destination). Specifies an address register in the postincrement addressing mode.

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Register Ay field — (always the source). Specifies an address register in the postincrement addressing mode.

CMP2 Compare Register Against Bounds CMP2

Operation: Compare Rn < lower-bound or
Rn > upper-bound
and Set Condition Codes

Assembler Syntax: CMP2 ⟨ea⟩,Rn

Attributes: Size = (Byte, Word, Long)

Description: Compares the value in Rn to each bound. The effective address contains the bounds pair: the lower bound followed by the upper bound. For signed comparisons, the arithmetically smaller value should be used as the lower bound. For unsigned comparisons, the logically smaller value should be the lower bound.

The size of the data and the bounds can be specified as byte, word, or long. If Rn is a data register and the operation size is byte or word, only the appropriate low-order part of Rn is checked. If Rn is an address register and the operation size is byte or word, the bounds operands are sign-extended to 32 bits and the resultant operands are compared to the full 32 bits of An.

If the upper bound equals the lower bound, the valid range is a single value.

NOTE

This instruction is identical to CHK2 except that it sets condition codes rather than taking an exception when the value in Rn is out of bounds.

Condition Codes:

X	N	Z	V	C
—	U	*	U	*

X Not affected.
N Undefined.
Z Set if Rn is equal to either bound. Cleared otherwise.
V Undefined.
C Set if Rn is out of bounds. Cleared otherwise.

Instruction Format:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2" colspan="2">SIZE</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td>D/A</td><td colspan="3">REGISTER</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Instruction Fields:

Size field — Specifies the size of the operation.
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the location of the bounds pair. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	—	—
(An)+	—	—			
−(An)	—	—			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

D/A field — Specifies whether an address register or data register is compared.
- 0 — Data register.
- 1 — Address register.

Register field — Specifies the address or data register that contains the value to be checked.

cpBcc Branch on Coprocessor Condition cpBcc

Operation: If cpcc true then scanPC + d → PC

Assembler Syntax: cpBcc ⟨label⟩

Attributes: Size = (Word, Long)

Description: If the specified coprocessor condition is true, program execution continues at location scanPC + displacement. The value of the scanPC is the address of the first displacement word. The displacement is a twos complement integer that represents the relative distance in bytes from the scan PC to the destination PC. The displacement can be either 16 bits or 32 bits. The coprocessor determines the specific condition from the condition field in the operation word. Refer to **9.4.1 ScanPC** for further information.

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			0	1	SIZE	COPROCESSOR CONDITION					
OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS															
WORD OR															
LONG WORD DISPLACEMENT															

Instruction Fields:

Cp-Id field — Identifies the coprocessor for this operation.

Size field — Specifies the size of the displacement.
- 0 — The displacement is 16 bits.
- 1 — The displacement is 32 bits.

Coprocessor Condition field — Specifies the coprocessor condition to be tested. This field is passed to the coprocessor which provides directives to the main processor for processing this instruction.

16-Bit Displacement field — The displacement value occupies 16 bits.

32-Bit Displacement field — The displacement value occupies 32 bits.

cpDBcc — Test Coprocessor Condition Decrement and Branch — cpDBcc

Operation: If cpcc false then (Dn − 1 → Dn; If Dn ≠ − 1 then scanPC + d → PC)

Assembler Syntax: cpDBcc Dn,⟨label⟩

Attributes: Size = (Word)

Description: If the specified coprocessor condition is true, execution continues with the next instruction. Otherwise, the low-order word in the specified data register is decremented by one. If the result is equal to −1, execution continues with the next instruction. If the result is not equal to −1, execution continues at the location indicated by the value of the scanPC plus the sign extended 16-bit displacement. The value of the scanPC is the address of the displacement word. The displacement is a twos complement integer that represents the relative distance in bytes from the scanPC to the destination PC. The coprocessor determines the specific condition from the condition word which follows the operation word. Refer to **9.4.1 ScanPC** for further information.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			0	0	1	0	0	1	REGISTER		
0	0	0	0	0	0	0	0	0	0	COPROCESSOR CONDITION					
OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS															
DISPLACEMENT (16 BIT)															

Instruction Fields:

Cp-Id field — Identifies the coprocessor for this operation.

Register field — Specifies the data register used as the counter.

Coprocessor Condition field — Specifies the coprocessor condition to be tested. This field is passed to the coprocessor which provides directives to the main processor for processing this instruction.

Displacement field — Specifies the distance of the branch (in bytes).

cpGEN Coprocessor General Function cpGEN

Operation: Pass Command Word to Coprocessor

Assembler Syntax: cpGEN ⟨parameters as defined by coprocessor⟩

Attributes: Unsized

Description: Transfers the command word that follows the operation word to the specified coprocessor. The coprocessor determines the specific operation from the command word. Usually a coprocessor defines specific instances of this instruction to provide its instruction set.

Condition Codes:

May be modified by coprocessor. Unchanged otherwise.

Instruction Format:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2" colspan="3">CP-ID</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td colspan="16">COPROCESSOR-DEPENDENT COMMAND WORD</td></tr>
<tr><td colspan="16">OPTIONAL EFFECTIVE ADDRESS OR COPROCESSOR-DEFINED EXTENSION WORDS</td></tr>
</table>

Instruction Fields:

Cp-Id field — Identifies the coprocessor for this operation.

Effective Address field — Specifies the location of any operand not resident in the coprocessor. The allowable addressing modes are determined by the operation to be performed.

Coprocessor Command field — Specifies the coprocessor operation to be performed. This word is passed to the coprocessor, which in turn provides directives to the main processor for processing this instruction.

cpRESTORE Coprocessor Restore Functions (Privileged Instruction) cpRESTORE

Operation: If supervisor state
then Restore Internal State of Coprocessor
else TRAP

Assembler Syntax: cpRESTORE ⟨ea⟩

Attributes: Unsized

Description: Restores the internal state of a coprocessor usually after it has been saved by a preceding cpSAVE instruction.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			1	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Field:
Cp-Id field — Identifies the coprocessor that is to be restored.
Effective Address field — Specifies the location where the internal state of the coprocessor is located. Only postincrement or control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

NOTE

If the format word returned by the coprocessor indicates "come again", pending interrupts are not serviced.

cpSAVE Coprocessor Save Function (Privileged Instruction) cpSAVE

Operation: If supervisor state
then Save Internal State of Coprocessor
else TRAP

Assembler Syntax: cpSAVE ⟨ea⟩

Attributes: Unsized

Description: Saves the internal state of a coprocessor in a format that can be restored by a cpRESTORE instruction.

Condition Codes:
Not affected.

Instruction Format:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2" colspan="3">CP-ID</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Instruction Fields:

Cp-Id field — Identifies the coprocessor for this operation.

Effective Address field — Specifies the location where the internal state of the coprocessor is to be saved. Only predecrement or control alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

cpScc Set On Coprocessor Condition cpScc

Operation: If cpcc true then 1s → Destination
else 0s → Destination

Assembler Syntax: cpScc ⟨ea⟩

Attributes: Size = (Byte)

Description: Tests the specified coprocessor condition code; if the condition is true, the byte specified by the effective address is set to TRUE (all ones), otherwise that byte is set to FALSE (all zeros). The coprocessor determines the specific condition from the condition word that follows the operation word.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			0	0	1	EFFECTIVE ADDRESS MODE			REGISTER		
0	0	0	0	0	0	0	0	0	0	COPROCESSOR CONDITION					
OPTIONAL EFFECTIVE ADDRESS OR COPROCESSOR-DEFINED EXTENSION WORDS															

Instruction Fields:

Cp-Id field — Identifies the coprocessor for this operation.

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Coprocessor Condition field — Specifies the coprocessor condition to be tested. This field is passed to the coprocessor, which in turn provides directives to the main processor for processing this instruction.

cpTRAPcc Trap On Coprocessor Condition ccTRAPcc

Operation: If cpcc true then TRAP

Assembler Syntax:
cpTRAPcc
cpTRAPcc #⟨data⟩

Attributes: Unsized or Size = (Word, Long)

Description: Tests the specified coprocessor condition code; if the selected coprocessor condition is true, the processor initiates a cpTRAPcc exception, vector number 7. The program counter value placed on the stack is the address of the next instruction. If the selected condition is not true, no operation is performed, and execution continues with the next instruction. The coprocessor determines the specific condition from the condition word that follows the operation word. Following the condition word is a user-defined data operand specified as immediate data to be used by the trap handler.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			0	0	1	1	1	1	OP-MODE		
0	0	0	0	0	0	0	0	0	0	COPROCESSOR CONDITION					
OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS															
OPTIONAL WORD															
OR LONG WORD OPERAND															

Instruction Fields:

Cp-Id field — Identifies the coprocessor for this operation.

Op-Mode field — Selects the instruction form.
- 010 — Instruction is followed by one operand word.
- 011 — Instruction is followed by two operand words.
- 100 — Instruction has no following operand words.

Coprocessor Condition field — Specifies the coprocessor condition to be tested. This field is passed to the coprocessor, which provides directives to the main processor for processing this instruction.

DBcc Test Condition, Decrement, and Branch DBcc

Operation: If condition false then (Dn $-$ 1 ➧ Dn;
If Dn $\neq$ $-$1 then PC + d ➧ PC)

Assembler Syntax: DBcc Dn,⟨label⟩

Attributes: Size = (Word)

Description: Controls a loop of instructions. The parameters are: a condition code, a data register (counter), and a displacement value. The instruction first tests the condition (for termination); if it is true, no operation is performed. If the termination condition is not true, the low-order 16 bits of the counter data register are decremented by one. If the result is -1, execution continues with the next instruction. If the result is not equal to -1, execution continues at the location indicated by the current value of the PC plus the sign-extended 16-bit displacement. The value in the PC is the address of the instruction word of the DBcc instruction plus two. The displacement is a twos complement integer that represents the relative distance in bytes from the current PC to the destination PC.

Condition code cc specifies one of the following conditions:

CC	carry clear	0100	$\overline{C}$	LS	low or same	0011	$C+Z$
CS	carry set	0101	C	LT	less than	1101	$N \bullet \overline{V} + \overline{N} \bullet V$
EQ	equal	0111	Z	MI	minus	1011	N
F	never equal	0001	0	NE	not equal	0110	$\overline{Z}$
GE	greater or equal	1100	$N \bullet V + \overline{N} \bullet \overline{V}$	PL	plus	1010	$\overline{N}$
GT	greater than	1110	$N \bullet V \bullet \overline{Z} + \overline{N} \bullet \overline{V} \bullet \overline{Z}$	T	always true	0000	1
HI	high	0010	$\overline{C} \bullet \overline{Z}$	VC	overflow clear	1000	$\overline{V}$
LE	less or equal	1111	$Z + N \bullet \overline{V} + \overline{N} \bullet V$	VS	overflow set	1001	V

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	CONDITION				1	1	0	0	1	REGISTER		
DISPLACEMENT (16 BITS)															

Instruction Fields:

Condition field — The binary code for one of the conditions listed in the table.
Register field — Specifies the data register used as the counter.
Displacement field — Specifies the number of bytes to branch.

NOTES:

1. The terminating condition is similar to the UNTIL loop clauses of high-level languages. For example: DBMI can be stated as "decrement and branch until minus".
2. Most assemblers accept DBRA for DBF for use when only a count terminates the loop (no condition is tested).
3. A program can enter a loop at the beginning or by branching to the trailing DBcc instruction. Entering the loop at the beginning is useful for indexed addressing modes and dynamically specified bit operations. In this case, the control index count must be one less than the desired number of loop executions. However, when entering a loop by branching directly to the trailing DBcc instruction, the control count should equal the loop execution count. In this case, if a zero count occurs, the DBcc instruction does not branch, and the main loop is not executed.

DIVS DIVSL

Signed Divide

Operation: Destination/Source ▶ Destination

Assembler Syntax:

DIVS.W ⟨ea⟩,Dn 32/16 ▶ 16r:16q
DIVS.L ⟨ea⟩,Dq 32/32 ▶ 32q
DIVS.L ⟨ea⟩,Dr:Dq 64/32 ▶ 32r:32q
DIVSL.L ⟨ea⟩,Dr:Dq 32/32 ▶ 32r:32q

Attributes: Size = (Word, Long)

Description: Divides the signed destination operand by the signed source operand and stores the signed result in the destination. The instruction uses one of four forms. The word form of the instruction divides a long word by a word. The result is a quotient in the lower word (least significant 16 bits) and the remainder is in the upper word (most significant 16 bits) of the result. The sign of the remainder is the same as the sign of the dividend.

The first long form divides a long word by a long word. The result is a long quotient; the remainder is discarded.

The second long form divides a quad word (in any two data registers) by a long word. The result is a long word quotient and a long word remainder.

The third long form divides a long word by a long word. The result is a long word quotient and a long word remainder.

Two special conditions may arise during the operation:

1. Division by zero causes a trap.
2. Overflow may be detected and set before the instruction completes. If the instruction detects an overflow, it sets the overflow condition code, and the operands are unaffected.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

X Not affected.
N Set if the quotient is negative. Cleared otherwise. Undefined if overflow or divide by zero occurs.
Z Set if the quotient is zero. Cleared otherwise. Undefined if overflow or divide by zero occurs.
V Set if division overflow occurs; undefined if divide by zero occurs. Cleared otherwise.
C Always cleared.

DIVS DIVSL — Signed Divide — DIVS DIVSL

Instruction Format (word form):

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="3" rowspan="2">REGISTER</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Instruction Fields:

Register field — Specifies any of the eight data registers. This field always specifies the destination operand.

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

NOTE

Overflow occurs if the quotient is larger than a 16-bit signed integer.

Instruction Format (long form):

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td>0</td><td colspan="3">REGISTER Dq</td><td>1</td><td>SIZE</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="3">REGISTER Dr</td></tr>
</table>

Instruction Fields:

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Register Dq field — Specifies a data register for the destination operand. The low-order 32 bits of the dividend comes from this register, and the 32-bit quotient is loaded into this register.

Size field — Selects a 32 or 64 bit division operation.
0 — 32-bit dividend is in Register Dq.
1 — 64-bit dividend is in Dr:Dq.

Register Dr field — After the division, this register contains the 32-bit remainder. If Dr and Dq are the same register, only the quotient is returned. If Size is 1, this field also specifies the data register that contains the high-order 32 bits of the dividend.

NOTE

Overflow occurs if the quotient is larger than a 32-bit signed integer.

DIVU DIVUL

Unsigned Divide

DIVU DIVUL

Operation: Destination/Source → Destination

Assembler Syntax:

DIVU.W ⟨ea⟩,Dn	32/16 → 16r:16q
DIVU.L ⟨ea⟩,Dq	32/32 → 32q
DIVU.L ⟨ea⟩,Dr:Dq	64/32 → 32r:32q
DIVUL.L ⟨ea⟩,Dr:Dq	32/32 → 32r:32q

Attributes: Size = (Word, Long)

Description: Divides the unsigned destination operand by the unsigned source operand and stores the unsigned result in the destination. The instruction uses one of four forms. The word form of the instruction divides a long word by a word. The result is a quotient in the lower word (least significant 16 bits) and the remainder is in the upper word (most significant 16 bits) of the result.

The first long form divides a long word by a long word. The result is a long quotient; the remainder is discarded.

The second long form divides a quad word (in any two data registers) by a long word. The result is a long word quotient and a long word remainder.

The third long form divides a long word by a long word. The result is a long word quotient and a long word remainder.

Two special conditions may arise during the operation:

1. Division by zero causes a trap.
2. Overflow may be detected and set before the instruction completes. If the instruction detects an overflow, it sets the overflow condition code, and the operands are unaffected.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

X Not affected.

N Set if the quotient is negative. Cleared otherwise. Undefined if overflow or divide by zero occurs.

Z Set if the quotient is zero. Cleared otherwise. Undefined if overflow or divide by zero occurs.

V Set if division overflow occurs; undefined if divide by zero occurs. Cleared otherwise.

C Always cleared.

DIVU
DIVUL

Unsigned Divide

DIVU
DIVUL

Instruction Format (word form):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	REGISTER			0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Register field — Specifies any of the eight data registers. This field always specifies the destination operand.

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

NOTE

Overflow occurs if the quotient is larger than a 16-bit signed integer.

DIVU DIVUL

Unsigned Divide

DIVU DIVUL

Instruction Format (long form):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dq			0	SIZE	0	0	0	0	0	0	0	REGISTER Dr		

Instruction Fields:

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
− (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Register Dq field — Specifies a data register for the destination operand. The low-order 32 bits of the dividend comes from this register, and the 32-bit quotient is loaded into this register.

Size field — Selects a 32- or 64-bit division operation.

0 — 32-bit dividend is in Register Dq.

1 — 64-bit dividend is in Dr:Dq.

Register Dr field — After the division, this register contains the 32-bit remainder. If Dr and Dq are the same register, only the quotient is returned. If Size is 1, this field also specifies the data register that contains the high-order 32 bits of the dividend.

NOTE

Overflow occurs if the quotient is larger than a 32-bit unsigned integer.

Operation: Source ⊕ Destination → Destination

Assembler Syntax: EOR Dn,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Performs an exclusive OR operation on the destination operand using the source operand and stores the result in the destination location. The size of the operation may be specified to be byte, word, or long. The source operand must be a data register. The destination operand is specified in the effective address field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format (word form):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Register field — Specifies any of the eight data registers.

Opmode field:

Byte	Word	Long	Operation
100	101	110	(⟨ea⟩) ⊕ (⟨Dn⟩) → ⟨ea⟩

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

NOTE

Memory to data register operations are not allowed. Most assemblers use EORI when the source is immediate data.

EORI Exclusive OR Immediate EORI

Operation: Immediate Data ⊕ Destination → Destination

Assembler Syntax: EORI #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Performs an exclusive OR operation on the destination operand using the immediate data and the destination operand and stores the result in the destination location. The size of the operation may be specified as byte, word, or long. The size of the immediate data matches the operation size.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
− (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Immediate field — (Data immediately following the instruction):
If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is next two immediate words.

EORI to CCR

Exclusive OR Immediate to Condition Code

EORI to CCR

Operation: Source ⊕ CCR → CCR

Assembler Syntax: EORI #⟨data⟩,CCR

Attributes: Size = (Byte)

Description: Performs an exclusive OR operation on the condition code register using the immediate operand and stores the result in the condition code register (low-order byte of the status register). All implemented bits of the condition code register are affected.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Changed if bit 4 of immediate operand is one. Unchanged otherwise.
N Changed if bit 3 of immediate operand is one. Unchanged otherwise.
Z Changed if bit 2 of immediate operand is one. Unchanged otherwise.
V Changed if bit 1 of immediate operand is one. Unchanged otherwise.
C Changed if bit 0 of immediate operand is one. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

EORI to SR

EORI to SR

Exclusive OR Immediate to the Status Register
(Privileged Instruction)

Operation: If supervisor state
then Source ⊕ SR → SR
else TRAP

Assembler Syntax: EORI #⟨data⟩,SR

Attributes: Size = (Word)

Description: Performs an exclusive OR operation on the contents of the status register using the immediate operand and stores the result in the status register. All implemented bits of the status register are affected.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Changed if bit 4 of immediate operand is one. Unchanged otherwise.
N Changed if bit 3 of immediate operand is one. Unchanged otherwise.
Z Changed if bit 2 of immediate operand is one. Unchanged otherwise.
V Changed if bit 1 of immediate operand is one. Unchanged otherwise.
C Changed if bit 0 of immediate operand is one. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0
WORD DATA (16 BITS)															

EXG Exchange Registers EXG

Operation: Rx ↔ Ry

Assembler Syntax:
EXG Dx,Dy
EXG Ax,Ay
EXG Dx,Ay
EXG Ay, Dx

Attributes: Size = (Long)

Description: Exchanges the contents of two 32-bit registers. The instruction performs three types of exchanges:

1. Exchange data registers.
2. Exchange address registers.
3. Exchange a data register and an address register.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER Rx			1	OPMODE					REGISTER Ry		

Instruction Fields:

Register Rx field — Specifies either a data register or an address register depending on the mode. If the exchange is between data and address registers, this field always specifies the data register.

Op-Mode field — Specifies the type of exchange:
01000 — Data registers.
01001 — Address registers.
10001 — Data register and address register.

Register Ry field — Specifies either a data register or an address register depending on the mode. If the exchange is between data and address registers, this field always specifies the address register.

EXT EXTB Sign Extend EXT EXTB

Operation: Destination Sign-extended → Destination

Assembler Syntax:

EXT.W Dn	extend byte to word
EXT.L Dn	extend word to long word
EXTB.L Dn	extend byte to long word

Attributes: Size = (Word, Long)

Description: Extends a byte in a data register to a word or a long word, or a word in a data register to a long word, by replicating the sign bit to the left. If the operation extends a byte to a word, bit [7] of the designated data register is copied to bits [15:8] of that data register. If the operation extends a word to a long word, bit [15] of the designated data register is copied to bits [31:16] of the data register. The EXTB form copies bit [7] of the designated register to bits [31:8] of the data register.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	OP-MODE			0	0	0	REGISTER		

Instruction Fields:

Op-Mode field — Specifies the size of the sign-extension operation:
- 010 — Sign-extend low-order byte of data register to word.
- 011 — Sign-extend low-order word of data register to long.
- 111 — Sign-extend low-order byte of data register to long.

Register field — Specifies the data register is to be sign-extended.

ILLEGAL Take Illegal Instruction Trap ILLEGAL

Operation: SSP − 2 → SSP; Vector Offset → (SSP);
SSP − 4 → SSP; PC → (SSP);
SSP − 2 → SSP; SR → (SSP);
Illegal Instruction Vector Address → PC

Assembler Syntax: ILLEGAL

Attributes: Unsized

Description: Forces an illegal instruction exception, vector number 4. All other illegal instruction bit patterns are reserved for future extension of the instruction set and should not be used to force an exception.

Condition Codes:
Not affected

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	1	1	1	1	0	0

JMP Jump JMP

Operation: Destination Address → PC

Assembler Syntax: JMP ⟨ea⟩

Attributes: Unsized

Description: Program execution continues at the effective address specified by the instruction. The addressing mode for the effective address must be a control addressing mode.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Effective Address field — Specifies the address of the next instruction. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	—	—
(An)+	—	—			
−(An)	—	—			
(d$_{16}$,An)	101	reg. number:An	(d$_{16}$,PC)	111	010
(d$_{8}$,An,Xn)	110	reg. number:An	(d$_{8}$,PC,Xn)	111	011
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

JSR Jump to Subroutine JSR

Operation: SP − 4 → Sp; PC → (SP)
Destination Address → PC

Assembler Syntax: JSR ⟨ea⟩

Attributes: Unsized

Description: Pushes the long word address of the instruction immediately following the JSR instruction onto the system stack. Program execution then continues at the address specified in the instruction.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		

Instruction Fields:

Effective Address field — Specifies the address of the next instruction. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
−(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

LEA Load Effective Address LEA

Operation: ⟨ea⟩ → An

Assembler Syntax: LEA ⟨ea⟩,An

Attributes: Size = (Long)

Description: Loads the effective address into the specified address register. All 32 bits of the address register are affected by this instruction.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	REGISTER			1	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Register field — Specifies the address register to be updated with the effective address.

Effective Address field — Specifies the address to be loaded into the address register. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
−(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

LINK Link and Allocate LINK

Operation: Sp − 4 ➧ Sp; An ➧ (SP);
SP ➧ An; SP + d ➧ SP

Assembler Syntax: LINK An, #⟨displacement⟩

Attributes: Size = (Word, Long)

Description: Pushes the contents of the specified address register onto the stack. Then loads the updated stack pointer into the address register. Finally, adds the displacement value to the stack pointer. For word size operation, the displacement is the sign-extended word following the operation word. For long size operation, the displacement is the long word following the operation word. The address register occupies one long word on the stack. The user should specify a negative displacement in order to allocate stack area.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	REGISTER		
WORD DISPLACEMENT															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	0	1	REGISTER		
HIGH-ORDER DISPLACEMENT															
LOW-ORDER DISPLACEMENT															

Instruction Fields:
Register field — Specifies the address register for the link.
Displacement field — Specifies the twos complement integer to be added to the stack pointer.

NOTE

LINK and UNLK can be used to maintain a linked list of local data and parameter areas on the stack for nested subroutine calls.

LSL, LSR Logical Shift LSL, LSR

Operation: Destination Shifted by ⟨count⟩ → Destination

Assembler Syntax:
LSd Dx,Dy
LSd #⟨data⟩,Dy
LSd ⟨ea⟩
where d is direction, L or R

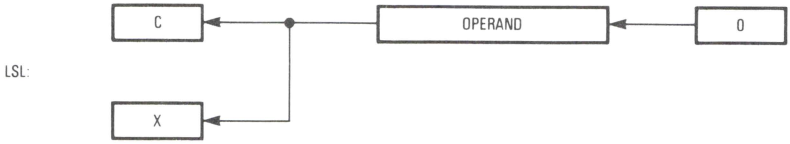
Attributes: Size = (Byte, Word, Long)

Description: Shifts the bits of the operand in the direction specified (L or R). The carry bit receives the last bit shifted out of the operand. The shift count for the shifting of a register is specified in two different ways:

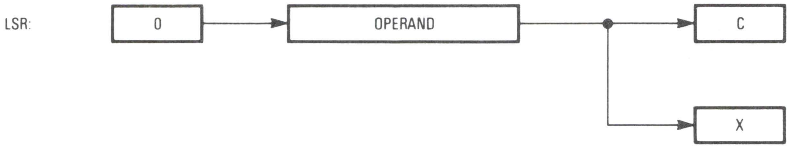
1. Immediate — The shift count (1-8) is specified in the instruction.
2. Register — The shift count is the value in the data register specified in the instruction modulo 64.

The size of the operation for register destinations may be specified as byte, word, or long. The contents of memory, ⟨ea⟩, can be shifted one bit only, and the operand size is restricted to a word.

The LSL instruction shifts the operand to the left the number of positions specified as the shift count. Bits shifted out of the high-order bit go to both the carry and the extend bits; zeros are shifted into the low-order bit.



The LSR instruction shifts the operand to the right the number of positions specified as the shift count. Bits shifted out of the low-order bit go to both the carry and the extend bits; zeros are shifted into the high-order bit.



Condition Codes:

X	N	Z	V	C
*	*	*	0	*

X Set according to the last bit shifted out of the operand. Unaffected for a shift count of zero.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Set according to the last bit shifted out of the operand. Cleared for a shift count of zero.

Instruction Format (Register Shifts):

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td colspan="3">COUNT/REGISTER</td><td>dr</td><td colspan="2">SIZE</td><td>i/r</td><td>0</td><td>1</td><td colspan="3">REGISTER</td></tr>
</table>

Instruction Field (Register Shifts):

Count/Register field:
  If i/r = 0, this field contains the shift count. The values 1-7 represent shifts of 1-7; value of 0 specifies a shift count of 8.
  If i/r = 1, the data register specified in this field contains the shift count (modulo 64).
dr field — Specifies the direction of the shift:
  0 — Shift right
  1 — Shift left
Size field — Specifies the size of the operation:
  00 — Byte operation
  01 — Word operation
  10 — Long operation
i/r field:
  If i/r = 0, specifies immediate shift count.
  If i/r = 1, specifies register shift count.
Register field — Specifies a data register to be shifted.

Instruction Format (Memory Shifts):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Memory Shifts):

dr field — Specifies the direction of the shift:
- 0 — Shift right
- 1 — Shift left

Effective Address field — Specifies the operand to be shifted. Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

MOVE Move Data from Source to Destination MOVE

Operation: Source → Destination

Assembler Syntax: MOVE ⟨ea⟩,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Moves the data at the source to the destination location, and sets the condition codes according to the data. The size of the operation may be specified as byte, word, or long.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	SIZE		DESTINATION						SOURCE					
				REGISTER			MODE			MODE			REGISTER		

Instruction Fields:

Size field — Specifies the size of the operand to be moved:
- 01 — Byte operation
- 11 — Word operation
- 10 — Long operation

Destination Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	—	—
(An)+	011	reg. number:An			
−(An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	—	—
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	—	—
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	—	—
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	—	—
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	—	—

Source Effective Address field — Specifies the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An*	001	reg. number:An	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	111	100
(An)+	011	reg. number:An			
−(An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

*For byte size operation, address register direct is not allowed.

NOTES:

1. Most assemblers use MOVEA when the destination is an address register.
2. MOVEQ can be used to move an immediate 8-bit value to a data register.

MOVEA

Move Address

Operation: Source → Destination

Assembler Syntax: MOVEA ⟨ea⟩,An

Attributes: Size = (Word, Long)

Description: Moves the contents of the source to the destination address register. The size of the operation is specified as word or long. Word-size source operands are sign-extended to 32-bit quantities.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	SIZE		DESTINATION REGISTER			0	0	1	SOURCE					
										MODE			REGISTER		

Instruction Fields:

Size field — Specifies the size of the operand to be moved:
- 11 — Word operation. The source operand is sign-extended to a long operand and all 32 bits are loaded into the address register.
- 10 — Long operation.

Destination Register field — Specifies the destination address register.

Effective Address field — Specifies the location of the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

MOVE from CCR

MOVE from CCR

Move from the Condition Code Register

Operation: CCR → Destination

Assembler Syntax: MOVE CCR,⟨ea⟩

Attributes: Size = (Word)

Description: Moves the condition code bits (zero extended to word size) to the destination location. The operand size is a word. Unimplemented bits are read as zeros.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	—	—
(An)+	011	reg. number:An			
−(An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	—	—
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	—	—
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	—	—
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	—	—
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	—	—

NOTE

MOVE from CCR is a word operation. ANDI, ORI, and EORI to CCR are byte operations.

MOVE to CCR

Move to Condition Codes

MOVE to CCR

Operation: Source → CCR

Assembler Syntax: MOVE ⟨ea⟩,CCR

Attributes: Size = (Word)

Description: Moves the low-order byte of the source operand to the condition code register. The upper byte of the source operand is ignored; the upper byte of the status register is not altered.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set to the value of bit 4 of the source operand.
N Set to the value of bit 3 of the source operand.
Z Set to the value of bit 2 of the source operand.
V Set to the value of bit 1 of the source operand.
C Set to the value of bit 0 of the source operand.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

MOVE to CCR — Move to Condition Codes — MOVE to CCR

Instruction Fields:

Effective Address field — Specifies the location of the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
− (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

NOTE

MOVE to CCR is a word operation. ANDI, ORI, and EORI to CCR are byte operations.

MOVE from SR

Move from the Status Register
(Privileged Instruction)

Operation: If supervisor state
then SR → Destination
else TRAP

Assembler Syntax: MOVE SR,⟨ea⟩

Attributes: Size = (Word)

Description: Moves the data in the status register to the destination location. The destination is word length. Unimplemented bits are read as zeros.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

NOTE

Use the MOVE from CCR instruction to access only the condition codes.

MOVE to SR

Move to the Status Register
(Priviledged Instruction)

Operation: If supervisor state
then Source → SR
else TRAP

Assembler Syntax: MOVE ⟨ea⟩,SR

Attributes: Size = (Word)

Description: Moves the data in the source operand to the status register. The source operand is a word and all implemented bits of the status register are affected.

Condition Codes:
Set according to the source operand.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:
Effective Address field — Specifies the location of the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

MOVE USP

Move User Stack Pointer
(Privileged Instruction)

Operation: If supervisor state
  then USP → An or An → USP
  else TRAP

Assembler Syntax: MOVE USP,An
MOVE An,USP

Attributes: Size = (Long)

Description: Moves the contents of the user stack pointer to or from the specified address register.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	dr	REGISTER		

Instruction Fields:
dr field — Specifies the direction of transfer:
  0 — Transfer the address register to the USP.
  1 — Transfer the USP to the address register.
Register field — Specifies the address register for the operation.

MOVEC Move Control Register (Privileged Instruction) MOVEC

Operation: If supervisor state
then Rc → Rn or Rn → Rc
else TRAP

Assembler Syntax:
MOVEC Rc,Rn
MOVEC Rn,Rc

Attributes: Size = (Long)

Description: Moves the contents of the specified control register (Rc) to the specified general register (Rn) or copies the contents of the specified general register to the specified control register. This is always a 32-bit transfer even though the control register may be implemented with fewer bits. Unimplemented bits are read as zeros.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	dr
A/D	REGISTER			CONTROL REGISTER											

Instruction Fields:

dr field — Specifies the direction of the transfer:
0 — Control register to general register.
1 — General register to control register.

A/D field — Specifies the type of general register:
0 — Data register.
1 — Address register.

Register field — Specifies the register number.

Control Register field — Specifies the control register.

Hex	Control Register
000	Source Function Code (SFC)
001	Destination Function Code (DFC)
002	Cache Control Register (CACR)
800	User Stack Pointer (USP)
801	Vector Base Register (VBR)
802	Cache Address Register (CAAR)
803	Master Stack Pointer (MSP)
804	Interrupt Stack Pointer (ISP)

Any other code causes an illegal instruction exception.

MOVEM Move Multiple Registers MOVEM

Operation: Registers ➧ Destination
Source ➧ Registers

Assembler Syntax: MOVEM register list,⟨ea⟩
MOVEM ⟨ea⟩,register list

Attributes: Size = (Word, Long)

Description: Moves the contents of selected registers to or from consecutive memory locations starting at the location specified by the effective address. A register is selected if the bit in the mask field corresponding to that register is set. The instruction size determines whether 16 or 32 bits of each register are transferred. In the case of a word transfer to either address or data registers, each word is sign-extended to 32 bits, and the resulting long word is loaded into the associated register.

Selecting the addressing mode also selects the mode of operation of the MOVEM instruction, and only the control modes, the predecrement mode, and the postincrement mode are valid. If the effective address is specified by one of the control modes, the registers are transferred starting at the specified address, and the address is incremented by the operand length (2 or 4) following each transfer. The order of the registers is from data register 0 to data register 7, then from address register 0 to address register 7.

If the effective address is specified by the predecrement mode, only a register-to-memory operation is allowed. The registers are stored starting at the specified address minus the operand length (2 or 4), and the address is decremented by the operand length following each transfer. The order of storing is from address register 7 to address register 0, then from data register 7 to data register 0. When the instruction has completed, the decremented address register contains the address of the last operand stored. In the MC68020, if the addressing register is also moved to memory, the value written is the decremented value.

If the effective address is specified by the postincrement mode, only a memory-to-register operation is allowed. The registers are loaded starting at the specified address; the address is incremented by the operand length (2 or 4) following each transfer. The order of loading is the same as that of control mode addressing. When the instruction has completed, the incremented address register contains the address of the last operand loaded plus the operand length. In the MC68020, if the addressing register is also loaded from memory, the value loaded is the value fetched plus the operand length.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										EFFECTIVE ADDRESS					
0	1	0	0	1	dr	0	0	1	SIZE	MODE			REGISTER		
REGISTER LIST MASK															

Instruction Field:

dr field — Specifies the direction of the transfer:

- 0 — Register to memory
- 1 — Memory to register

Size field — Specifies the size of the registers being transferred:

- 0 — Word transfer
- 1 — Long transfer

Effective Address field — Specifies the memory address for the operation. For register-to-memory transfers, only control alterable addressing modes or the predecrement addressing mode are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	—	—
(An) +	—	—			
− (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	—	—
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	—	—
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	—	—
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	—	—
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	—	—

For memory-to-register transfers, only control addressing modes or the postincrement addressing mode are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	—	—	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	—	—
(An)+	011	reg. number:An			
−(An)	—	—			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

Register List Mask field — Specifies the registers to be transferred. The low-order bit corresponds to the first register to be transferred; the high-order bit corresponds to the last register to be transferred. Thus, both for control modes and for the postincrement mode addresses, the mask correspondence is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

For the predecrement mode addresses, the mask correspondence is reversed:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

NOTE

An extra read bus cycle occurs for memory operands. This accesses an operand at one address higher than the last register image required.

MOVEP Move Peripheral Data MOVEP

Operation: Source → Destination

Assembler Syntax:
MOVEP Dx,(d,Ay)
MOVEP (d,Ay),Dx

Attributes: Size = (Word, Long)

Description: Moves data between a data register and alternate bytes within the address space (typically assigned to a peripheral), starting at the location specified and incrementing by two. This instruction is designed for 8-bit peripherals on a 16-bit data bus. The high-order byte of the data register is transferred first and the low-order byte is transferred last. The memory address is specified in the address register indirect plus 16-bit displacement addressing mode. If the address is even, all the transfers are to or from the high-order half of the data bus; if the address is odd, all the transfers are to or from the low-order half of the data bus. The instruction also accesses alternate bytes on an 8- or 32-bit bus.

Example: Long transfer to/from an even address.

Byte Organization in Register

31 24	23 16	15 8	7 0
HI-ORDER	MID-UPPER	MID-LOWER	LOW-ORDER

Byte Organization in Memory (Low Address at Top)

15 8	7 0
HI-ORDER	
MID-UPPER	
MID-LOWER	
LOW-ORDER	

Example: Word transfer to/from an odd address

Byte Organization in Register

31 24	23 16	15 8	7 0
		HI-ORDER	LOW-ORDER

Byte Organization in Memory (Low Address at Top)

15 8	7 0
	HI-ORDER
	LOW-ORDER

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	DATA REGISTER			OPMODE			0	0	1	ADDRESS REGISTER		
DISPLACEMENT (16 BITS)															

Instruction Fields:

Data Register field — Specifies the data register for the instruction.

Opmode field — Specifies the direction and size of the operation:

- 100 — Transfer word from memory to register.
- 101 — Transfer long from memory to register.
- 110 — Transfer word from register to memory.
- 111 — Transfer long from register to memory.

Address Register field — Specifies the address register which is used in the address register indirect plus displacement addressing mode.

Displacement field — Specifies the displacement used in the operand address.

MOVEQ Move Quick MOVEQ

Operation: Immediate Data ▶ Destination

Assembler Syntax: MOVEQ #⟨data⟩,Dn

Attributes: Size = (Long)

Description: Moves a byte of immediate data to a 32-bit data register. The data in an 8-bit field within the operation word is sign-extended to a long operand in the data register as it is transferred.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	REGISTER			0	DATA							

Instruction Fields:

Register field — Specifies the data register to be loaded.
Data field — Eight bits of data, which are sign-extended to a long operand.

MOVES   Move Address Space (Privileged Instruction)   MOVES

Operation: If supervisor state
then Rn → Destination [DFC] or Source [SFC] → Rn
else TRAP

Assembler Syntax:
MOVES Rn,⟨ea⟩
MOVES ⟨ea⟩,Rn

Attributes: Size = (Byte, Word, Long)

Description: Moves the byte, word, or long operand from the specified general register to a location within the address space specified by the destination function code (DFC) register; or, moves the byte, word, or long operand from a location within the address space specified by the source function code (SFC) register to the specified general register.

If the destination is a data register, the source operand replaces the corresponding low-order bits of that data register, depending on the size of the operation. If the destination is an address register, the source operand is sign-extended to 32 bits and then loaded into that address register.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
A/D	REGISTER			dr	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields:
Size field — Specifies the size of the operation:
00 — Byte operation
01 — Word operation
10 — Long operation

Effective Address field — Specifies the source or destination location within the alternate address space. Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

A/D field — Specifies the type of general register:
0 — Data register
1 — Address register

Register field — Specifies the register number.

dr field — Specifies the direction of the transfer:
0 — From ⟨ea⟩ to general register
1 — From general register to ⟨ea⟩

NOTE

For either of the two following examples with the same address register as both source and destination

MOVES.x An,(An)+
MOVES.x An,−(An)

the value stored is undefined. The current implementations of the MC68010 and MC68020 store the incremented or decremented value of An.

MULS Signed Multiply MULS

Operation: Source * Destination ▶ Destination

Assembler Syntax:

MULS.W ⟨ea⟩,Dn	16x16 ▶ 32
MULS.L ⟨ea⟩,Dl	32x32 ▶ 32
MULS.L ⟨ea⟩,Dh:Dl	32 x 32 ▶ 64

Attributes: Size = (Word, Long)

Description: Multiplies two signed operands yielding a signed result. This instruction has a word operand form and a long word operand form.

In the word form, the multiplier and multiplicand are both word operands, and the result is a long word operand. A register operand is the low-order word; the upper word of the register is ignored. All 32 bits of the product are saved in the destination data register.

In the long form, the multiplier and multiplicand are both long word operands, and the result is either a long word or a quad word. The long word result is the low-order 32 bits of the quad word result; the high-order 32 bits of the product are discarded.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if overflow. Cleared otherwise.
C Always cleared.

NOTE

Overflow (V=1) can occur only when multiplying 32-bit operands to yield a 32-bit result. Overflow occurs if the high-order 32 bits of the quad word product are not the sign extension of the low-order 32 bits.

Instruction Format (word form):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER			1	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

MULS Signed Multiply MULS

Instruction Fields:

Register field — Specifies a data register as the destination.

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Instruction Format (long form):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	REGISTER Dq			1	SIZE	0	0	0	0	0	0	0	REGISTER Dr		

Instruction Fields:

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Register Dl field — Specifies a data register for the destination operand. The 32-bit multiplicand comes from this register, and the low-order 32 bits of the product are loaded into this register.

Size field — Selects a 32- or 64-bit product.
0 — 32-bit product to be returned to Register Dl.
1 — 64-bit product to be returned to Dh:Dl.

Register Dh field — If Size is 1, specifies the data register into which the high-order 32 bits of the product are loaded. If Dh = Dl and Size is 1, the results of the operation are undefined. Otherwise, this field is unused.

MULU Unsigned Multiply MULU

Operation: Source * Destination ➧ Destination

Assembler Syntax:

MULU.W ⟨ea⟩,Dn 16x16 ➧ 32
MULU.L ⟨ea⟩,Dl 32x32 ➧ 32
MULU.L ⟨ea⟩,Dh:Dl 32x32 ➧64

Attributes: Size = (Word, Long)

Description: Multiplies two unsigned operands yielding an unsigned result. This instruction has a word operand form and a long word operand form.

In the word form, the multiplier and multiplicand are both word operands, and the result is a long word operand. A register operand is the low-order word; the upper word of the register is ignored. All 32 bits of the product are saved in the destination data register.

In the long form, the multiplier and multiplicand are both long word operands, and the result is either a long word or a quad word. The long word result is the low-order 32 bits of the quad word result; the high-order 32 bits of the product are discarded.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if overflow. Cleared otherwise.
C Always cleared.

NOTE

Overflow (V = 1) can occur only when multiplying 32-bit operands to yield a 32-bit result. Overflow occurs if any of the high-order 32 bits of the quad word product are not equal to zero.

Instruction Format (word form):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER			0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Register field — Specifies a data register as the destination.

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Instruction Format (long form):

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td>0</td><td colspan="3">REGISTER Dl</td><td>0</td><td>SIZE</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="3">REGISTER Dh</td></tr>
</table>

Instruction Fields:

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

Register Dl field — Specifies a data register for the destination operand. The 32-bit multiplicand comes from this register, and the low-order 32 bits of the product are loaded into this register.

Size field — Selects a 32- or 64-bit product.

0 — 32-bit product to be returned to Register Dl.

1 — 64-bit product to be returned to Dh:Dl.

Register Dh field — If Size is 1, specifies the data register into which the high-order 32 bits of the product are loaded. If Dh = Dl and Size is 1, the results of the operation are undefined. Otherwise, this field is unused.

NBCD Negate Decimal with Extend NBCD

Operation: $0 - (\text{Destination}_{10}) - X \rightarrow$ Destination

Assembler Syntax: NBCD ⟨ea⟩

Attributes: Size = (Byte)

Description: Subtracts the destination operand and the extend bit from zero. The operation is performed using binary coded decimal arithmetic. The packed BCD result is saved in the destination location. This instruction produces the tens complement of the destination if the extend bit is zero, or the nines complement if the extend bit is one. This is a byte operation only.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

X Set the same as the carry bit.
N Undefined.
Z Cleared if the result is non-zero. Unchanged otherwise.
V Undefined.
C Set if a decimal borrow occurs. Cleared otherwise.

NOTE

Normally the Z condition code bit is set via programming before the start of the operation. This allows successful tests for zero results upon completion of multiple precision operations.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

NEG Negate NEG

Operation: 0 − (Destination) ▶ Destination

Assembler Syntax: NEG ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the destination operand from zero and stores the result in the destination location. The size of the operation is specified as byte, word, or long.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set the same as the carry bit.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Cleared if the result is zero. Set otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Size field — Specifies the size of the operation.
  00 — Byte operation
  01 — Word operation
  10 — Long operation

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

NEGX Negate with Extend NEGX

Operation: 0 – (Destination) – X ▶ Destination

Assembler Syntax: NEGX ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the destination operand and the extend bit from zero. Stores the result in the destination location. The size of the operation is specified as byte, word, or long.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set the same as the carry bit.
N Set if the result is negative. Cleared otherwise.
Z Cleared if the result is nonzero. Unchanged otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Set if a borrow occurs. Cleared otherwise.

NOTE

Normally the Z condition code bit is set via programming before the start of the operation. This allows successful tests for zero results upon completion of multiple precision operations.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Size field — Specifies the size of the operation.
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

NOP None NOP

Operation: None

Assembler Syntax: NOP

Attributes: Unsized

Description: Performs no operation. The processor state, other than the program counter, is unaffected. Execution continues with the instruction following the NOP instruction. The NOP instruction does not begin execution until all pending bus cycles are completed. This synchronizes the pipeline, and prevents instruction overlap. See **3.7 PIPELINE SYNCHRONIZATION WITH THE NOP INSTRUCTION** for additional information.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1

NOT Logical Complement NOT

Operation: ~ Destination → Destination

Assembler Syntax: NOT ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Calculates the ones complement of the destination operand and stores the result in the destination location. The size of the operation is specified as byte, word, or long.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Size field — Specifies the size of the operation.
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
− (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

OR Inclusive OR Logical OR

Operation: Source V Destination → Destination

Assembler Syntax:
OR ⟨ea⟩,Dn
OR Dn,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Performs an inclusive OR operation on the source operand and the destination operand and stores the result in the destination location. The size of the operation is specified as byte, word, or long. The contents of an address register may not be used as an operand.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Register field — Specifies any of the eight data registers.
Opmode field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩) V (⟨Dn⟩) → ⟨Dn⟩
100	101	110	(⟨Dn⟩) V (⟨ea⟩) → ⟨ea⟩

Effective Address field — If the location specified is a source operand, only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	100	reg. number:An
(d$_{16}$,An)	101	reg. number:An
(d$_{8}$,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d$_{16}$,PC)	111	010
(d$_{8}$,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

If the location specified is a destination operand, only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
–(An)	100	reg. number:An
(d$_{16}$,An)	101	reg. number:An
(d$_{8}$,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d$_{16}$,PC)	—	—
(d$_{8}$,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

NOTES:

1. If the destination is a data register, it must be specified using the destination Dn mode, not the destination ⟨ea⟩ mode.
2. Most assemblers use ORI when the source is immediate data.

ORI Inclusive OR ORI

Operation: Immediate Data V Destination → Destination

Assembler Syntax: ORI #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Performs an inclusive OR operation on the immediate data and the destination operand and stores the result in the destination location. The size of the operation is specified as byte, word, or long. The size of the immediate data matches the operation size.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="2" rowspan="2">SIZE</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td colspan="8">WORD DATA (16 BITS)</td><td colspan="8">BYTE DATA (8 BITS)</td></tr>
<tr><td colspan="16">LONG DATA (32 BITS)</td></tr>
</table>

Instruction Fields:

Size field — Specifies the size of the operation.
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Immediate field — (Data immediately following the instruction):

If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is the next two immediate words.

ORI to CCR

Inclusive OR Immediate to Condition Codes

Operation: Source V CCR → CCR

Assembler Syntax: ORI #⟨data⟩,CCR

Attributes: Size = (Byte)

Description: Performs an inclusive OR operation on the immediate operand and the condition codes and stores the result in the condition code register (low-order byte of the status register). All implemented bits of the condition code register are affected.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set if bit 4 of immediate operand is one. Unchanged otherwise.
N Set if bit 3 of immediate operand is one. Unchanged otherwise.
Z Set if bit 2 of immediate operand is one. Unchanged otherwise.
V Set if bit 1 of immediate operand is one. Unchanged otherwise.
C Set if bit 0 of immediate operand is one. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

ORI to SR

Inclusive OR Immediate to the Status Register (Privileged Instruction)

Operation: If supervisor state
then Source V SR → SR
else TRAP

Assembler Syntax: ORI #⟨data⟩,SR

Attributes: Size = (Word)

Description: Performs an inclusive OR operation of the immediate operand and the contents of the status register and stores the result in the status register. All implemented bits of the status register are affected.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set if bit 4 of immediate operand is one. Unchanged otherwise.
N Set if bit 3 of immediate operand is one. Unchanged otherwise.
Z Set if bit 2 of immediate operand is one. Unchanged otherwise.
V Set if bit 1 of immediate operand is one. Unchanged otherwise.
C Set if bit 0 of immediate operand is one. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
WORD DATA (16 BITS)															

PACK Pack PACK

Operation: Source (Unpacked BCD) + adjustment ➧ Destination (Packed BCD)

Assembler Syntax:
PACK −(Ax),−(Ay),#⟨adjustment⟩
PACK Dx,Dy,#⟨adjustment⟩

Attributes: Unsized

Description: Adjusts and packs the low four bits of each of two bytes into a single byte.

When both operands are data registers, the adjustment is added to the value contained in the source register. Bits [11:8] and [3:0] of the intermediate result are concatenated and placed in bits [7:0] of the destination register. The remainder of the destination register is unaffected.

Source (Dx):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	a	b	c	d	x	x	x	x	e	f	g	h

Add Adjustment Word:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16-BIT EXTENSION															

Resulting in:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x′	x′	x′	x′	a′	b′	c′	d′	x′	x′	x′	x′	e′	f′	g′	h′

Destination (Dy):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
u	u	u	u	u	u	u	u	a′	b′	c′	d′	e′	f′	g′	h′

When the predecrement addressing mode is specified, two bytes from the source are fetched and concatenated. The adjustment word is added to the concatenated bytes. Bits [3:0] of each byte are extracted. These eight bits are concatenated to form a new byte which is then written to the destination.

Source (Ax):

7	6	5	4	3	2	1	0
x	x	x	x	a	b	c	d
x	x	x	x	e	f	g	h

Concatenated Word:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	a	b	c	d	x	x	x	x	e	f	g	h

Add Adjustment Word:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16-BIT EXTENSION															

Destination (Ay):

7	6	5	4	3	2	1	0
a′	b′	c′	d′	e′	f′	g′	h′

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	REGISTER Dy/Ay			1	0	1	0	0	R/M	REGISTER Dx/Ax		
16-BIT EXTENSION: ADJUSTMENT															

Instruction Fields:

Register Dy/Ay field — Specifies the destination register.
- If R/M = 0, specifies a data register.
- If R/M = 1, specifies an address register in the predecrement addressing mode.

R/M field — Specifies the operand addressing mode.
- 0 — The operation is data register to data register.
- 1 — The operation is memory to memory.

Register Dx/Ax field — Specifies the source register.
- If R/M = 0, specifies a data register.
- If R/M = 1, specifies an address register in the predecrement addressing mode.

Adjustment field — Immediate data word that is added to the source operand. This word is zero to pack ASCII or EBCDIC codes. Other values can be used for other codes.

PEA Push Effective Address PEA

Operation: Sp − 4 ➧ SP; ⟨ea⟩ ➧ (SP)

Assembler Syntax: PEA ⟨ea⟩

Attributes: Size = (Long)

Description: Computes the effective address and pushes it onto the stack. The effective address is a long word address.

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Effective Address field — Specifies the address to be pushed onto the stack. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
− (An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

RESET — Reset External Devices (Privileged Instruction) — RESET

Operation: If supervisor state
  then Assert $\overline{\text{RESET}}$ Line
  else TRAP

Assembler Syntax: RESET

Attributes: Unsized

Description: Asserts the $\overline{\text{RESET}}$ signal for 512 clock periods, resetting all external devices. The processor state, other than the program counter, is unaffected and execution continues with the next instruction.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0

ROL, ROR   Rotate (Without Extend)   ROL, ROR

Operation: Destination Rotated by ⟨count⟩ → Destination

Assembler Syntax:
ROd Dx,Dy
ROd #⟨data⟩,Dy
ROd ⟨ea⟩
where d is direction, L or R

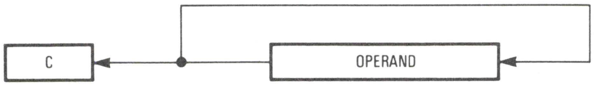
Attributes: Size = (Byte, Word, Long)

Description: Rotates the bits of the operand in the direction specified (L or R). The extend bit is not included in the rotation. The rotate count for the rotation of a register is specified in either of two ways:

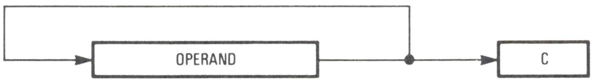
1. Immediate — The rotate count (1-8) is specified in the instruction.
2. Register — The rotate count is the value in the data register specified in the instruction, modulo 64.

The size of the operation for register destinations is specified as byte, word, or long. The contents of memory, ⟨ea⟩; can be rotated one bit only, and operand size is restricted to a word.

The ROL instruction rotates the bits of the operand to the left; the rotate count determines the number of bit positions rotated. Bits rotated out of the high-order bit go to the carry bit and also back into the low-order bit.



The ROR instruction rotates the bits of the operand to the right; the rotate count determines the number of bit positions rotated. Bits rotated out of the low-order bit go to the carry bit and also back into the high-order bit.



Condition Codes:

X	N	Z	V	C
—	*	*	0	*

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Set according to the last bit rotated out of the operand. Cleared when the rotate count is zero.

Instruction Format (Register Rotate):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	COUNT/ REGISTER			dr	SIZE		i/r	1	1	REGISTER		

Instruction Fields (Register Rotate):

Count/Register field:
- If i/r = 0, this field contains the rotate count. The values 1-7 represent counts of 1-7, and 0 specifies a count of 8.
- If i/r = 1, this field specifies a data register that contains the rotate count (modulo 64).

dr field — Specifies the direction of the rotate:
- 0 — Rotate right
- 1 — Rotate left

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

i/r field — Specifies the rotate count location:
- If i/r = 0, immediate rotate count.
- If i/r = 1, register rotate count.

Register field — Specifies a data register to be rotated.

Instruction Format (Memory Rotate):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Memory Rotate):

dr field — Specifies the direction of the rotate:

0 — Rotate right
1 — Rotate left

Effective Address field — Specifies the operand to be rotated. Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Operation: Destination Rotated with X by ⟨count⟩ ➧ Destination

Assembler Syntax:
ROXd Dx,Dy
ROXd #⟨data⟩,Dy
ROXd ⟨ea⟩
where d is direction, L or R

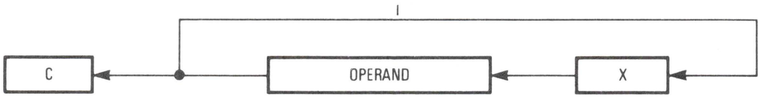
Attributes: Size = (Byte, Word, Long)

Description: Rotates the bits of the operand in the direction specified (L or R). The extend bit is included in the rotation. The rotate count for the rotation of a register is specified in either of two ways:

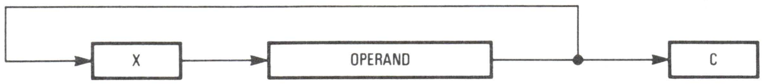
1. Immediate — The rotate count (1-8) is specified in the instruction.
2. Register — The rotate count is the value in the data register specified in the instruction, modulo 64.

The size of the operation for register destinations is specified as byte, word, or long. The contents of memory, ⟨ea⟩, can be rotated one bit only, and operand size is restricted to a word.

The ROXL instruction rotates the bits of the operand to the left; the rotate count determines the number of bit positions rotated. Bits rotated out of the high-order bit go to the carry bit and the extend bit; the previous value of the extend bit rotates into the low-order bit.



The ROXR instruction rotates the bits of the operand to the right; the rotate count determines the number of bit positions rotated. Bits rotated out of the low-order bit go to the carry bit and the extend bit; the previous value of the extend bit rotates into the high-order bit.



Condition Codes:

X	N	Z	V	C
*	*	*	0	*

X Set to the value of the last bit rotated out of the operand. Unaffected when the rotate count is zero.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Set according to the last bit rotated out of the operand. When the rotate count is zero, set to the value of the extend bit.

Instruction Format (Register Rotate):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	COUNT/ REGISTER			dr	SIZE		i/r	1	0	REGISTER		

Instruction Fields (Register Rotate):

Count/Register field:
- If i/r = 0, this field contains the rotate count. The values 1-7 represent counts of 1-7, and 0 specifies a count of 8.
- If i/r = 1, this field specifies a data register that contains the rotate count (modulo 64).

dr field — Specifies the direction of the rotate:
- 0 — Rotate right
- 1 — Rotate left

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

i/r field — Specifies the rotate count location:
- If i/r = 0, immediate rotate count.
- If i/r = 1, register rotate count.

Register field — Specifies a data register to be rotated.

Instruction Format (Memory Rotate):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Memory Rotate):

dr field — Specifies the direction of the rotate:

0 — Rotate right

1 — Rotate left

Effective Address field — Specifies the operand to be rotated. Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

RTD Return and Deallocate RTD

Operation: (SP) → PC; SP + 4 + d → SP

Assembler Syntax: RTD #⟨displacement⟩

Attributes: Unsized

Description: Pulls the program counter value from the stack and adds the sign-extended 16-bit displacement value to the stack pointer. The previous program counter value is lost.

Condition Codes:
Not affected.

Instruction Format:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td colspan="16">DISPLACEMENT (16 BITS)</td></tr>
</table>

Instruction Field:

Displacement field — Specifies the twos complement integer to be sign extended and added to the stack pointer.

RTM Return from Module RTM

Operation: Reload Saved Module State from Stack

Assembler Syntax: RTM Rn

Attributes: Unsized

Description: A previously saved module state is reloaded from the top of stack. After the module state is retrieved from the top of the stack, the caller's stack pointer is incremented by the argument count value in the module state.

Condition Codes:
Set according to the content of the word on the stack.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	D/A	REGISTER		

Instruction Fields:

D/A field — Specifies whether the module data pointer is in a data or an address register.
0 — the register is a data register
1 — the register is an address register

Register field — Specifies the register number for the module data area pointer which is to be restored from the saved module state. If the register specified in A7 (SP), the updated value of the register reflects the stack pointer operations, and the saved module data area pointer is lost.

RTE — Return from Exception (Privileged Instruction) — RTE

Operation:

```
If supervisor state
   then (SP) → SR; SP + 2 → SP; (SP) → PC;
   SP + 4 → SP;
   restore state and deallocate stack according to (SP)
   else TRAP
```

Assembler Syntax: RTE

Attributes: Unsized

Description: Loads the processor state information stored in the exception stack frame located at the top of the stack into the processor. The instruction examines the stack format field in the format/offset word to determine how much information must be restored.

Condition Codes:

Set according to the condition code bits in the status register value restored from the stack.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1

Format/Offset word (in stack frame):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FORMAT				0	0	VECTOR OFFSET									

Format Field of Format/Offset Word:

Contains the format code, which implies the stack frame size (including the format/offset word).

0000 — Short Format, removes four words. Loads the status register and the program counter from the stack frame.

0001 — Throwaway Format, removes four words. Loads the status register from the stack frame and switches to the active system stack. Continues the instruction using the active system stack.

0010 — Instruction Error Format, removes six words. Loads the status register and the program counter from the stack frame and discards the other words.

1000 — MC68010 Long Format. The MC68020 takes a format error exception.

1001 — Coprocessor Mid-Instruction Format, removes 10 words. Resumes execution of coprocessor instruction.

1010 — MC68020 Short Format, removes 16 words and resumes instruction execution.

1011 — MC68020 Long Format, removes 46 words and resumes instruction execution.

Any other value in this field causes the processor to take a format error exception.

RTR Return and Restore Condition Codes RTR

Operation: (SP) → CCR; SP + 2 → SP;
(SP) → PC; SP + 4 → SP

Assembler Syntax: RTR

Attributes: Unsized

Description: Pulls the condition code and program counter values from the stack. The previous condition codes and program counter values are lost. The supervisor portion of the status register is unaffected.

Condition Codes:
Set to the condition codes from the stack.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	1

RTS Return from Subroutine RTS

Operation: (SP) ▶ PC; SP + 4 ▶ SP

Assembler Syntax: RTS

Attributes: Unsized

Description: Pulls the program counter value from the stack. The previous program counter value is lost.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1

SBCD Subtract Decimal with Extend SBCD

Operation: Destination$_{10}$ − Source$_{10}$ − X $\rightarrow$ Destination

Assembler Syntax:
SBCD Dx,Dy
SBCD −(Ax),−(Ay)

Attributes: Size = (Byte)

Description: Subtracts the source operand and the extend bit from the destination operand and stores the result in the destination location. The subtraction is performed using binary coded decimal arithmetic; the operands are packed BCD numbers. The instruction has two modes:

1. Data register to data register: The data registers specified in the instruction contain the operands.
2. Memory to memory: The address registers specified in the instruction access the operands from memory using the predecrement addressing mode.

This operation is a byte operation only.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

X Set the same as the carry bit.
N Undefined.
Z Cleared if the result is nonzero. Unchanged otherwise.
V Undefined.
C Set if a borrow (decimal) is generated. Cleared otherwise.

NOTE

Normally the Z condition code bit is set via programming before the start of an operation. This allows successful tests for zero results upon completion of multiple-precision operations.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	REGISTER Ry			1	0	0	0	0	R/M	REGISTER Rx		

Instruction Fields:

Register Dy/Ay field — Specifies the destination register.
  If R/M = 0, specifies a data register.
  If R/M = 1, specifies an address register for the predecrement addressing mode.
R/M field — Specifies the operand addressing mode:
  0 — The operation is data register to data register.
  1 — The operation is memory to memory.
Register Dx/Ax field — Specifies the source register:
  If R/M = 0, specifies a data register.
  If R/M = 1, specifies an address register for the predecrement addressing mode.

Scc — Set According to Condition — Scc

Operation: If Condition True
then 1s ➧ Destination
else 0s ➧ Destination

Assembler Syntax: Scc ⟨ea⟩

Attributes: Size = (Byte)

Description: Tests the specified condition code; if the condition is true, sets the byte specified by the effective address to TRUE (all ones). Otherwise, sets that byte to FALSE (all zeros). Condition code cc specifies one of the following conditions:

CC	carry clear	0100	$\overline{C}$	LS	low or same	0011	$C + Z$
CS	carry set	0101	C	LT	less than	1101	$N \bullet \overline{V} + \overline{N} \bullet V$
EQ	equal	0111	Z	MI	minus	1011	N
F	never true	0001	0	NE	not equal	0110	$\overline{Z}$
GE	greater or equal	1100	$N \bullet V + \overline{N} \bullet \overline{V}$	PL	plus	1010	$\overline{N}1$
GT	greater than	1110	$N \bullet V \bullet \overline{Z} + \overline{N} \bullet \overline{V} \bullet \overline{Z}$	T	always true	0000	$\overline{V}$
HI	high	0010	$\overline{C} \bullet \overline{Z}$	VC	overflow clear	1000	V
LE	less or equal	1111	$Z + N \bullet \overline{V} + \overline{N} \bullet V$	VS	overflow set	1001	

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	CONDITION				1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Condition field — The binary code for one of the conditions listed in the table.

Effective Address field — Specifies the location in which the true/false byte is to be stored. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
− (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

NOTE:

A subsequent NEG.B instruction with the same effective address can be used to change the Scc result from TRUE or FALSE to the equivalent arithmetic value (TRUE = 1, FALSE = 0).

STOP Load Status Register and Stop (Privileged Instruction) STOP

Operation: If supervisor state
then Immediate Data → SR; STOP
else TRAP

Assembler Syntax: STOP #⟨data⟩

Attributes: Unsized

Description: Moves the immediate operand into the status register (both user and supervisor portions), advances the program counter to point to the next instruction, and stops the fetching and executing of instructions. A trace, interrupt, or reset exception causes the processor to resume instruction execution. A trace exception occurs if instruction tracing is enabled (T0 = 1, T1 = 0) when the STOP instruction begins execution. If an interrupt request is asserted with a priority higher than the priority level set by the new status register value, an interrupt exception occurs; otherwise, the interrupt request is ignored. External reset always initiates reset exception processing.

Condition Codes:

Set according to the immediate operand.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0
IMMEDIATE DATA															

Instruction Fields:

Immediate field — Specifies the data to be loaded into the status register.

SUB Subtract SUB

Operation: Destination – Source ➧ Destination

Assembler Syntax:
SUB ⟨ea⟩,Dn
SUB Dn,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the source operand from the destination operand and stores the result in the destination. The size of the operation is specified as byte, word, or long. The mode of the instruction indicates which operand is the source, which is the destination, and which is the operand size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set to the value of the carry bit.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a borrow is generated. Cleared otherwise.

Instruction Format:

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
						EFFECTIVE ADDRESS	
1	0	0	1	REGISTER	OPMODE	MODE	REGISTER

Instruction Fields:

Register field — Specifies any of the eight data registers.

Opmode field:

Byte	**Word**	**Long**	**Operation**
000	001	010	(⟨Dn⟩) – (⟨ea⟩) ➧ ⟨Dn⟩
100	101	110	(⟨ea⟩) – (⟨Dn⟩) ➧ ⟨ea⟩

Effective Address field — Determines the addressing mode. If the location specified is a source operand, all addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*For byte size operation, address register direct is not allowed.

If the location specified is a destination operand, only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

NOTES:

1. If the destination is a data register, it must be specified as a destination Dn address, not as a destination ⟨ea⟩ address.
2. Most assemblers use SUBA when the destination is an address register, and SUBI or SUBQ when the source is immediate data.

Operation: Destination − Source → Destination

Assembler Syntax: SUBA ⟨ea⟩,An

Attributes: Size = (Word, Long)

Description: Subtracts the source operand from the destination address register and stores the result in the address register. The size of the operation is specified as word or long. Word size source operands are sign extended to 32-bit quantities prior to the subtraction.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Opmode Field:

Word	Long	Operation
011	111	(⟨An⟩) − (⟨ea⟩) → ⟨An⟩

Instruction Fields:

Register field — Specifies the destination, any of the eight address registers.

Opmode field — Specifies the size of the operation:

011 — Word operation. The source operand is sign extended to a long operand and the operation is performed on the address register using all 32 bits.

111 — Long operation.

Effective Address field — Specifies the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An	001	reg. number:An	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	111	100
(An)+	011	reg. number:An			
−(An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

SUBI **Subtract Immediate SUBI** SUBI

Operation: Destination − Immediate Data → Destination

Assembler
Syntax: SUBI #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the immediate data from the destination operand and stores the result in the destination location. The size of the operation is specified as byte, word, or long. The size of the immediate data matches the operation size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set to the value of the carry bit.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Set if a borrow occurs. Cleared otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	SIZE		EFFECTIVE ADDRESS MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Instruction Fields:

Size field — Specifies the size of the operation.
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

Immediate field — (Data immediately following the instruction)
If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is the next two immediate words.

SUBQ Subtract Quick SUBQ

Operation: Destination − Immediate Data → Destination

Assembler Syntax: SUBQ #⟨data⟩,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the immediate data (1-8) from the destination operand. The size of the operation is specified as byte, word, or long. Only word and long operations are allowed with address registers, and the condition codes are not affected. When subtracting from address registers, the entire destination address register is used, regardless of the operation size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set to the value of the carry bit.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Set if a borrow occurs. Cleared otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	DATA			1	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Data field — Three bits of immediate data; 1-7 represent immediate values of 1-7, and 0 represents 8.

Size field — Specifies the size of the operation:

00 — Byte operation
01 — Word operation
10 — Long operation

Effective Address field — Specifies the destination location. Only alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
− (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Word and Long Only

SUBX Subtract with Extend SUBX

Operation: Destination − Source − X → Destination

Assembler Syntax:
SUBX Dx,Dy
SUBX −(Ax),−(Ay)

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the source operand and the extend bit from the destination operand and stores the result in the destination location. The instruction has two modes:

1. Data register to data register: The data registers specified in the instruction contain the operands.
2. Memory to memory: The address registers specified in the instruction access the operands from memory using the predecrement addressing mode.

The size of the operand is specified as byte, word, or long.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set to the value of the carry bit.
N Set if the result is negative. Cleared otherwise.
Z Cleared if the result is nonzero. Unchanged otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Set if a carry occurs. Cleared otherwise.

NOTE

Normally the Z condition code bit is set via programming before the start of an operation. This allows successful tests for zero results upon completion of multiple-precision operations.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	REGISTER Ry			1	SIZE		0	0	R/M	REGISTER Rx		

Instruction Fields:

Register Dy/Ay field — Specifies the destination register:
  If R/M = 0, specifies a data register.
  If R/M = 1, specifies an address register for the predecrement addressing mode.
Size field — Specifies the size of the operation:
  00 — Byte operation
  01 — Word operation
  10 — Long operation
R/M field — Specifies the operand addressing mode:
  0 — The operation is data register to data register.
  1 — The operation is memory to memory.
Register Dx/Ax field — Specifies the source register:
  If R/M = 0, specifies a data register.
  If R/M = 1, specifies an address register for the predecrement addressing mode.

SWAP Swap Register Halves SWAP

Operation: Register [31:16] ↔ Register [15:0]

Assembler Syntax: SWAP Dn

Attributes: Size = (Word)

Description: Exchange the 16-bit words (halves) of a data register.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the 32-bit result is set. Cleared otherwise.
Z Set if the 32-bit result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	0	REGISTER		

Instruction Fields:

Register field — Specifies the data register to swap.

TAS Test and Set an Operand TAS

Operation: Destination Tested ➧ Condition Codes; 1 ➧ bit 7 of Destination

Assembler Syntax: TAS ⟨ea⟩

Attributes: Size = (Byte)

Description: Tests and sets the byte operand addressed by the effective address field. The instruction tests the current value of the operand and sets the N and Z condition bits appropriately. TAS also sets the high-order bit of the operand. The operation uses a read-modify-write memory cycle that completes the operation without interruption. This instruction supports use of a flag or semaphore to coordinate several processors.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the operand is currently set. Cleared otherwise.
Z Set if the operand was zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Effective Address field — Specifies the location of the tested operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

TRAP Trap TRAP

Operation: SSP − 2 ▶ SSP; Format/Offset ▶ (SSP);
SSP − 4 ▶ SSP; PC ▶ (SSP); SSP − 2 ▶ SSP;
SR ▶ (SSP); Vector Address ▶ PC

Assembler Syntax: TRAP #⟨vector⟩

Attributes: Unsized

Description: Causes a TRAP #⟨vector⟩ exception. The instruction adds the immediate operand (vector) of the instruction to 32 to obtain the vector number. The range of vector values is 0-15, which provides 16 vectors.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	0	VECTOR			

Instruction Fields:
Vector field — Specifies the trap vector to be taken.

TRAPcc Trap on Condition TRAPcc

Operation: If cc then TRAP

Assembler Syntax:
TRAPcc
TRAPcc.W #⟨data⟩
TRAPcc.L #⟨data⟩

Attributes: Unsized or Size = (Word, Long)

Description: If the specified condition is true, causes a TRAPcc exception. The vector number is 7. The processor pushes the address of the next instruction word (currently in the program counter) onto the stack. If the condition is not true, the processor performs no operation, and execution continues with the next instruction. The immediate data operand should be placed in the next word(s) following the operation word and is available to the trap handler. Condition code cc specifies one of the following conditions.

CC	carry clear	0100	$\overline{C}$	LS	low or same	0011	$C+Z$
CS	carry set	0101	C	LT	less than	1101	$N\bullet\overline{V}+\overline{N}\bullet V$
EQ	equal	0111	Z	MI	minus	1011	N
F	never true	0001	0	NE	not equal	0110	$\overline{Z}$
GE	greater or equal	1100	$N\bullet V+\overline{N}\bullet\overline{V}$	PL	plus	1010	$\overline{N}$
GT	greater than	1110	$N\bullet V\bullet\overline{Z}+\overline{N}\bullet\overline{V}\bullet\overline{Z}$	T	always true	0000	1
HI	high	0010	$\overline{C}\bullet\overline{Z}$	VC	overflow clear	1000	$\overline{V}$
LE	less or equal	1111	$Z+N\bullet\overline{V}+\overline{N}\bullet V$	VS	overflow set	1001	V

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	CONDITION				1	1	1	1	1	OPMODE		
OPTIONAL WORD															
OR LONG WORD															

Instruction Fields:

Condition field — The binary code for one of the conditions listed in the table.

Opmode field — Selects the instruction form.
- 010 — Instruction is followed by word-size operand.
- 011 — Instruction is followed by long-word-size operand.
- 100 — Instruction has no operand.

TRAPV Trap on Overflow TRAPV

Operation: If V then TRAP

Assembler Syntax: TRAPV

Attributes: Unsized

Description: If the overflow condition is set, causes a TRAPV exception (vector number 7). If the overflow condition is not set, the processor performs no operation and execution continues with the next instruction.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	0

TST Test an Operand TST

Operation: Destination Tested ▶ Condition Codes

Assembler Syntax: TST ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Compares the operand with zero and sets the condition codes according to the results of the test. The size of the operation is specified as byte, word, or long.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the operand is negative. Cleared otherwise.
Z Set if the operand is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. If the operation size is word or long, all addressing modes are allowed. If the operation size is byte, only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An	001	reg. number:An*	(xxx).L	111	001
(An)	010	reg. number:An	#⟨data⟩	—	—
(An) +	011	reg. number:An			
− (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

*Word or long word operation only.

UNLK Unlink UNLK

Operation: An → SP; (SP) → An; SP + 4 → SP

Assembler Syntax: UNLK An

Attributes: Unsized

Description: Loads the stack pointer from the specified address register then loads the address register with the long word pulled from the top of the stack.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	REGISTER		

Instruction Fields:
Register field — Specifies the address register for the instruction.

UNPK Unpack BCD UNPK

Operation: Source (Packed BCD) + adjustment ▶ Destination (Unpacked BCD)

Assembler Syntax:
UNPACK −(Ax),−(Ay),#⟨adjustment⟩
UNPK Dx,Dy,#⟨adjustment⟩

Attributes: Unsized

Description: Places the two BCD digits in the source operand byte into the lower nibbles of two bytes, and places zero bits in the upper nibbles of both bytes. Adds the adjustment value to this unpacked value. Condition codes are not altered.

When both operands are data registers, the instruction unpacks the source register contents, adds the extension word, and places the result in the destination register. The high word of the destination register is unaffected.

Source (Dx):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
u	u	u	u	u	u	u	u	a	b	c	d	e	f	g	h

Intermediate Expansion:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	a	b	c	d	0	0	0	0	e	f	g	h

Add Adjustment Word:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="16">16-BIT EXTENSION</td></tr>
</table>

Destination (Dy):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
v	v	v	v	a′	b′	c′	d′	w	w	w	w	e′	f′	g′	h′

When the specified addressing mode is predecrement, the instruction extracts two BCD digits from a byte at the source address. After unpacking the digits and adding the adjustment word, the instruction writes the two bytes to the destination address.

Source (Ax):

7	6	5	4	3	2	1	0
a	b	c	d	e	f	g	h

UNPK Unpack BCD UNPK

Intermediate Expansion:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	a	b	c	d	0	0	0	0	e	f	g	h

Add Adjustment Word:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="16">16-BIT EXTENSION</td></tr>
</table>

Destination (Ay):

<table>
<tr><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>v</td><td>v</td><td>v</td><td>v</td><td>a'</td><td>b'</td><td>c'</td><td>d'</td></tr>
<tr><td>w</td><td>w</td><td>w</td><td>w</td><td>e'</td><td>f'</td><td>g'</td><td>h'</td></tr>
<tr><td colspan="8"></td></tr>
</table>

Condition Codes:

Not affected.

Instruction Format:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="3">REGISTER Dy/Ay</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>R/M</td><td colspan="3">REGISTER Dx/Ax</td></tr>
<tr><td colspan="16">16-BIT EXTENSION: ADJUSTMENT</td></tr>
</table>

Instruction Fields:

Register Dy/Ay field — Specifies the destination register.
- If R/M = 0, specifies a data register.
- If R/M = 1, specifies an address register in the predecrement addressing mode.

R/M field — Specifies the operand addressing mode.
- 0 — The operation is data register to data register.
- 1 — The operation is memory to memory.

Register Dx/Ax field — Specifies the data register.
- If R/M = 0, specifies a data register.
- If R/M = 1, specifies an address register in the predecrement addressing mode.

Adjustment field — Immediate data word that is added to the source operand. Appropriate constants can be used as the adjustment, to translate from BCD to the desired code. The constant used for ASCII is $3030; for EBCDIC, $F0F0.

3.4 USING THE CAS AND CAS2 INSTRUCTIONS

The CAS instruction compares the value in a memory location with the value in a data register, and copies a second data register into the memory location if the compared values are equal. This provides a means of updating system counters, history information, and globally shared pointers. The instruction uses a indivisible read-modify-write cycle; after CAS reads the memory location, no other instruction can change that location before CAS has written the new value. This provides security in single processor systems, in multitasking environments, and in multiprocessor environments. In a single processor system, the operation is protected from instructions of an interrupt routine. In a multitasking environment, no other task can interfere with writing the new value of a system variable. In a multiprocessor environment, the other processors must wait until the CAS instruction completes before accessing a global pointer.

The following code fragment shows a routine to maintain a count, in location SYS_CNTR, of the executions of an operation that may be performed by any process or processor in a system. The routine obtains the current value of the count in register D0 and stores the new count value in register D1. The CAS instruction copies the new count into SYS_CNTR if it is valid. But if another user has incremented the count between the time the count was stored and the read-modify-write cycle of the CAS instruction, the write portion of the cycle copies the new count in SYS_CNTR into D0, and the routine branches to try again. The following code sequence guarantees that SYS_CNTR is correctly incremented.

```
           MOVE.W    SYS_CNTR,D0        get the old value of the counter
INC_LOOP   MOVE.W    D0,D1              make a copy of it
           ADDQ.W    #1,D1              and increment it
           CAS.W     D0,D1,SYS_CNTR     if counter value is still the same, update it
           BNE       INC_LOOP           if not, try again
```

The CAS and CAS2 instructions together allow safe operations in the manipulation of system linked lists. Controlling a single location, HEAD in the example, manages a last-in-first-out linked list. If the list is empty, HEAD contains the NULL pointer (0); otherwise, HEAD contains the address of the element most recently added to the list. The following code fragment illustrates the code for inserting an element. The MOVE instructions load the address in location HEAD into D0 and into the NEXT pointer in the element being inserted, and the address of the new element into D1. The CAS instruction stores the address of the inserted element into location HEAD if the address in HEAD remains unaltered. If HEAD contains a new address, the instruction loads the new address into D0 and branches to the second MOVE instruction to try again. Figure 3-3 shows the insertion operation.

The CAS2 instruction is similar to the CAS instruction except that it performs two comparisons and updates two variables when the results of the comparisons are equal. If the results of both comparisons are equal, CAS2 copies new values into the destination addresses. If the result of either comparison is not equal, the instruction copies the values in the destination addresses into the compare operands.

The next code fragment shows the use of a CAS2 instruction to delete an element from a linked list. The first LEA instruction loads the effective address of HEAD into A0. The MOVE instruction loads the address in pointer HEAD into D0. The TST instruction checks for an empty list, and the BEQ instruction branches to a routine at label SDEMPTY if the list is

```
SINSERT                                     ALLOCATE NEW ENTRY, ADDRESS IN A1
              MOVE.L      HEAD,D0           MOVE HEAD POINTER VALUE TO D0
SILOOP        MOVE.L      D0,(NEXT,A1)      ESTABLISH FORWARD LINK IN NEW ENTRY
              MOVE.L      A1,D1             MOVE NEW ENTRY POINTER VALUE TO D1
              CAS.L       D0,D1,HEAD        IF WE STILL POINT TO TOP OF STACK, UPDATE THE HEAD POINTER
              BNE         SILOOP            IF NOT, TRY AGAIN
```

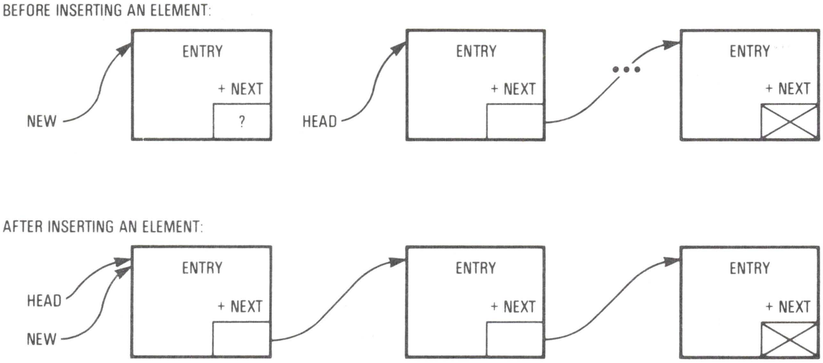


Figure 3-3. Linked List Insertion

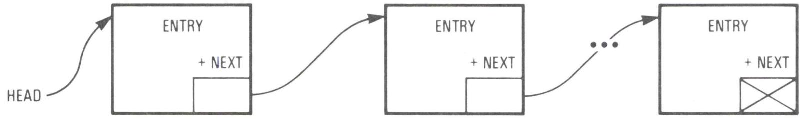
empty. Otherwise, a second LEA instruction loads the address of the NEXT pointer in the newest element on the list into A1, and the following MOVE instruction loads the pointer contents into D1. The CAS2 instruction compares the address of the newest structure to the value in HEAD and the address in D1 to the pointer in the address in A1. If no element has been inserted or deleted by another routine while this routine has been executing, the results of these comparisons are equal, and the CAS2 instruction stores the new value into location HEAD. If an element has been inserted or deleted, the CAS2 instruction loads the new address in location HEAD into D0, and the BNE instruction branches to the TST instruction to try again. Figure 3-4 shows the deletion of an element from a linked list.

The CAS2 instruction can also be used to correctly maintain a first-in-first-out doubly-linked list. A doubly-linked list needs two controlled locations, LIST_PUT and LIST_GET, which contain pointers to the last element inserted in the list and the next to be removed, respectively. If the list is empty, both pointers are NULL (0).

The following code fragment illustrates the insertion of an element in a doubly-linked list. The first two instructions load the effective addresses of LIST_PUT and LIST_GET into registers A0 and A1, respectively. The next instruction moves the address of the new element into register D2. Another MOVE instruction moves the address in LIST_PUT into register D0. At label DILOOP, a TST instruction tests the value in D0, and the BEQ instruction branches to the MOVE instruction when D0 is equal to zero. Assuming the list is empty, this MOVE instruction is executed next; it moves the zero in D0 into the NEXT and LAST pointers of the new element. Then the CAS2 instruction moves the address of the new element into both LIST_PUT and LIST_GET, assuming that both of these pointers still contain zero. If not, the BNE instruction branches to the TST instruction at label DILOOP

```
SDELETE
          LEA       HEAD,A0                 LOAD ADDRESS OF HEAD POINTER INTO A0
          MOVE.L    (A0),D0                 MOVE VALUE OF HEAD POINTER INTO D0
SDLOOP    TST.L     D0                      CHECK FOR NULL HEAD POINTER
          BEQ       SDEMPTY                 IF EMPTY, NOTHING TO DELETE
          LEA       (NEXT,D0),A1            LOAD ADDRESS OF FORWARD LINK INTO A1
          MOVE.L    (A1),D1                 PUT FORWARD LINK VALUE IN D1
          CAS2.L    D0:D1,D1:D1,(A0):(A1)   IF STILL POINT TO ENTRY TO BE DELETED, THEN UPDATE HEAD AND FORWARD POINTERS
          BNE       SDLOOP                  IF NOT, TRY AGAIN
SDEMPTY                                     SUCCESSFUL DELETION, ADDRESS OF DELETED ENTRY IN D0 (MAY BE NULL)
```

BEFORE DELETING AN ELEMENT:



AFTER DELETING AN ELEMENT:

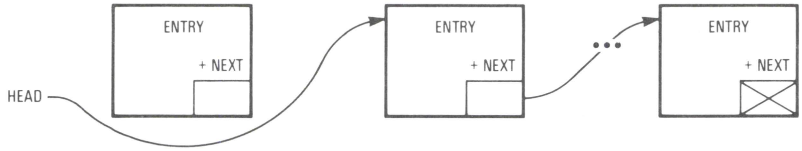


Figure 3-4. Linked List Deletion

to try again. This time, the BEQ instruction does not branch, and the following MOVE instruction moves the address in D0 to the NEXT pointer of the new element. The CLR instruction clears register D1 to zero, and the MOVE instruction moves the zero into the LAST pointer of the new element. The LEA instruction loads the address of the LAST pointer of the most recently inserted element into register A1. Assuming the LIST_PUT pointer and the pointer in A1 have not been changed, the CAS2 instruction stores the address of the new element into these pointers. Figure 3-5 shows insertion of a new element into a doubly-linked list.

The code fragment to delete an element from a doubly-linked list is similar. The first two instructions load the effective addresses of pointers LIST_PUT and LIST_GET into registers A0 and A1, respectively. The MOVE instruction at label DDLOOP moves the LIST_GET pointer into register D1. The BEQ instruction that follows branches out of the routine when the pointer is zero. The MOVE instruction moves the LAST pointer of the element to be deleted into register D2. Assuming this is not the last element in the list, the Z condition code is not set, and the branch to label DDEMPTY does not occur. The LEA instruction loads the address of the NEXT pointer of the element at the address in D2 into register A2. The next instruction, a CLR instruction, clears register D0 to zero. The CAS2 instruction compares the address in D1 to the LIST_GET pointer and to the address in register A2. If the pointers have not been updated, the CAS2 instruction loads the address in D2 into the LIST_GET pointer and zero into the address in register A2.

When the list contains only one element, the routine branches to the CAS2 instruction at label DDEMPTY after moving a zero pointer value into D2. This instruction checks the

```
DINSERT                                   (ALLOCATE NEW LIST ENTRY, LOAD ADDRESS INTO A2)
          LEA       LIST_PUT,A0           LOAD ADDRESS OF HEAD POINTER INTO A0
          LEA       LIST_GET,A1           LOAD ADDRESS OF TAIL POINTER INTO A1
          MOVE.L    A2,D2                 LOAD NEW ENTRY POINTER INTO D2
          MOVE.L    (A0),D0               LOAD POINTER TO HEAD ENTRY INTO D0
DILOOP    TST.L     D0                    IS HEAD POINTER NULL (0 ENTRIES IN LIST)?
          BEQ       DIEMPTY               IF SO, WE NEED ONLY TO ESTABLISH POINTERS
          MOVE.L    D0,(NEXT,A2)          PUT HEAD POINTER INTO FORWARD POINTER OF NEW ENTRY
          CLR.L     D1                    PUT NULL POINTER VALUE IN D1
          MOVE.L    D1,(LAST,A2)          PUT NULL POINTER IN BACKWARD POINTER OF NEW ENTRY
          LEA       (LAST,D0),A1          LOAD BACKWARD POINTER OF OLD HEAD ENTRY INTO A1
          CAS2.L    D0:D1,D2:D2,(A0):(A1) IF WE STILL POINT TO OLD HEAD ENTRY, UPDATE POINTERS
          BNE       DILOOP                IF NOT, TRY AGAIN
          BRA       DIDONE
DIEMPTY   MOVE.L    D0,(NEXT,A2)          PUT NULL POINTER IN FORWARD POINTER OF NEW ENTRY
          MOVE.L    D0,(LAST,A2)          PUT NULL POINTER IN BACKWARD POINTER OF NEW ENTRY
          CAS2.L    D0:D0,D2:D2,(A0):(A1) IF WE STILL HAVE NO ENTRIES, SET BOTH POINTERS TO THIS ENTRY
          BNE       DILOOP                IF NOT, TRY AGAIN
DIDONE                                    SUCCESSFUL LIST ENTRY INSERTION
```

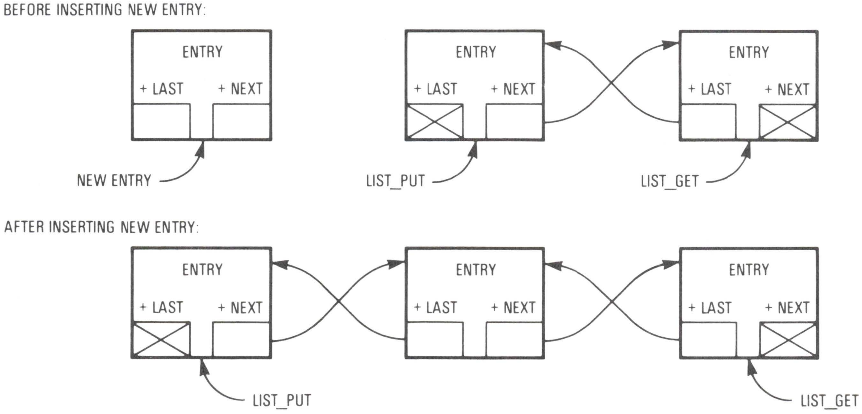


Figure 3-5. Doubly-Linked List Insertion

addresses in LIST_PUT and LIST_GET to verify that no other routine has inserted another element or deleted the last element. Then the instruction moves zero into both pointers, and the list is empty. Figure 3-6 shows the deletion of an element of a doubly-linked list.

3.5 NESTED SUBROUTINE CALLS

The LINK instruction pushes an address onto the stack, saves the stack address at which the address is stored, and reserves an area of the stack. Using this instruction in a series of subroutine calls results in a linked list of stack frames.

The UNLK instruction removes a stack frame from the end of the list by loading an address into the stack pointer and pulling the value at that address from the stack. When the operand

```
DDELETE
          LEA      LIST_PUT,A0              GET ADDRESS OF HEAD POINTER IN A0
          LEA      LIST_GET,A1              GET ADDRESS OF TAIL POINTER IN A1
DDLOOP    MOVE.L   (A1),D1                  MOVE TAIL POINTER INTO D1
          BEQ      DDDONE                   IF NO LIST, QUIT
          MOVE.L   (LAST,D1),D2             PUT BACKWARD POINTER IN D2
          BEQ      DDEMPTY                  IF ONLY ONE ELEMENT, UPDATE POINTERS
          LEA      (NEXT,D2),A2             PUT ADDRESS OF FORWARD POINTER IN A2
          CLR.L    D0                       PUT NULL POINTER VALUE IN D0
          CAS2.L   D1:D1,D2:D0,(A1):(A2)    IF BOTH POINTERS STILL POINT TO THIS ENTRY, UPDATE THEM
          BNE      DDLOOP                   IF NOT, TRY AGAIN
          BRA      DDDONE
DDEMPTY   CAS2.L   D1:D1,D2:D2,(A1):(A0)    IF STILL FIRST ENTRY, SET HEAD AND TAIL POINTERS TO NULL
          BNE      DDLOOP                   IF NOT, TRY AGAIN
DDDONE                                      SUCCESSFUL ENTRY DELETION, ADDRESS OF DELETED ENTRY IN D1 (MAY BE NULL)
```

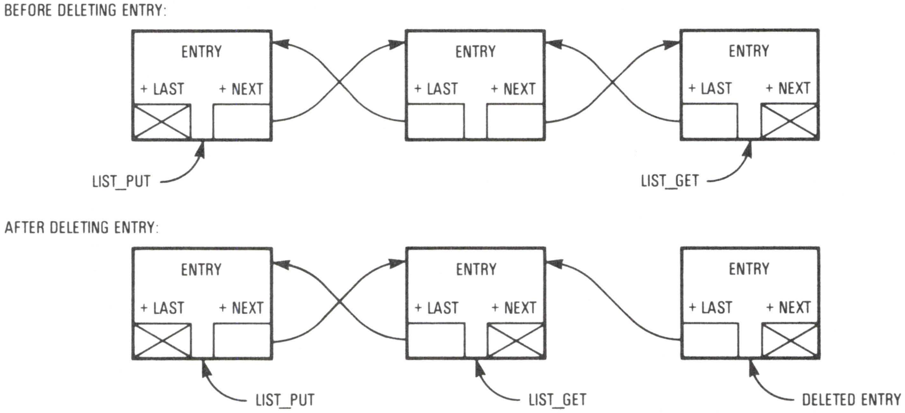


Figure 3-6. Doubly-Linked List Deletion

of the instruction is the address of the link address at the bottom of a stack frame, the effect is to remove the stack frame from the stack and from the linked list.

3.6 BIT FIELD INSTRUCTIONS

One of the data types provided by the MC68020 is the bit field, consisting of as many as 32 consecutive bits. A bit field is defined by an offset from an effective address and a width value. The offset is a value in the range of -2^{31} through $2^{31}-1$ from the most significant bit (bit 7) at the effective address. The width is a positive number, 1 through 32. The most significant bit of a bit field is bit 0; the bits number in a direction opposite to the bits of an integer.

The instruction set includes eight instructions that have bit field operands. The insert bit field (BFINS) instruction inserts a bit field stored in a register into a bit field. The extract bit field signed (BFEXTS) instruction loads a bit field into the least significant bits of a register and extends the sign to the left, filling the register. The extract bit field unsigned (BFEXTU) also loads a bit field, but zero fills the unused portion of the destination register.

The set bit field (BFSET) sets all the bits of a field to ones. The clear bit field (BFCLR) instruction clears a field. The change bit field (BFCHG) instruction complements all the bits in a bit field. These three instructions all test the previous value of the bit field, setting the condition codes accordingly. The test bit field (BFTST) tests the value in the field, setting the condition codes appropriately without altering the bit field. The find first one in bit field (BFFFO) instruction scans a bit field from bit 0 to the right until it finds a bit set to one and loads the bit offset of the first set bit into the specified data register. If no bits in the field are set, the field offset plus the field width is loaded into the register.

An important application of bit field instructions is the manipulation of the exponent field in a floating-point number. In the IEEE standard format, the most significant bit is the sign bit of the mantissa. The exponent value begins at the next most significant bit position; the exponent field does not begin on a byte boundary. The extract bit field instruction (BFEXTU) and the test bit field instruction (BFTST) are the most useful for this application, but other bit field instructions can also be used.

Programming of input and output operations to peripherals requires testing, setting, and inserting of bit fields in the control registers of the peripherals. This is another application for bit field instructions. However, control register locations are not memory locations; therefore, it is not always possible to insert or extract bit fields of a register without affecting other fields within the register.

Another widely used application for bit field instructions is bit-mapped graphics. Because byte boundaries are ignored in these areas of memory, the field definitions used with bit field instructions are very helpful.

3.7 PIPELINE SYNCHRONIZATION WITH THE NOP INSTRUCTION

Although the no operation (NOP) instruction performs no visible operation, it serves an important purpose. It forces synchronization of the instruction pipeline by waiting for all pending bus cycles to complete. All previous instructions complete execution before the NOP begins.

3.8 CONDITION CODES

The five least significant bits of the status register (condition codes) are set or cleared by many instructions to indicate the results of the instructions. The following paragraphs include:

1) a table that lists every instruction and how it sets the codes, and
2) another table that lists the mnemonics for conditional tests and their meanings in terms of condition code bits.

The condition codes were developed to meet two criteria:

- Consistency — across instructions, uses, and instances
- Meaningful Results — no change unless it provides useful information

Consistency across instructions means that all instructions that are special cases of more general instructions affect the condition codes in the same way. Consistency across instances means that all instances of an instruction affect the condition codes in the same way. Consistency across uses means that conditional instructions test the condition codes similarly and provide the same results whether the condition codes are set by a compare, test, or move instruction.

3.8.1 Condition Code Computation

Most operations take a source operand and a destination operand, compute, and store the result in the destination location. Single-operand operations take a destination operand, compute, and store the result in the destination location. Table 3-12 lists each instruction and how it sets the condition codes.

Table 3-12. Condition Code Computations (Sheet 1 of 2)

Operations	X	N	Z	V	C	Special Definition
ABCD	*	U	?	U	?	C = Decimal Carry $Z = Z \wedge \overline{Rm} \wedge \ldots \wedge \overline{R0}$
ADD, ADDI, ADDQ	*	*	*	?	?	$V = Sm \wedge Dm \wedge \overline{Rm} \vee \overline{Sm} \wedge \overline{Dm} \wedge Rm$ $C = Sm \wedge Dm \vee \overline{Rm} \wedge Dm \vee Sm \wedge \overline{Rm}$
ADDX	*	*	?	?	?	$V = Sm \wedge Dm \wedge \overline{Rm} \vee \overline{Sm} \wedge \overline{Dm} \wedge Rm$ $C = Sm \wedge Dm \vee \overline{Rm} \wedge Dm \vee Sm \wedge \overline{Rm}$ $Z = Z \wedge \overline{Rm} \wedge \ldots \wedge \overline{R0}$
AND, ANDI, EOR, EORI, MOVEQ, MOVE, OR, ORI, CLR, EXT, NOT, TAS, TST	—	*	*	0	0	
CHK	—	*	U	U	U	
CHK2, CMP2	—	U	?	U	?	$Z = (R = LB) \vee (R = UB)$ $C = (LB < = UB) \wedge (IR < LB) \vee (R > UB)) \vee (UB < LB) \wedge (R > UB) \wedge (R < LB)$
SUB, SUBI, SUBQ	*	*	*	?	?	$V = \overline{Sm} \wedge Dm \wedge \overline{Rm} \vee Sm \wedge \overline{Dm} \wedge Rm$ $C = Sm \wedge \overline{Dm} \vee Rm \wedge \overline{Dm} \vee Sm \wedge Rm$
SUBX	*	*	?	?	?	$V = \overline{Sm} \wedge Dm \wedge \overline{Rm} \vee Sm \wedge \overline{Dm} \wedge Rm$ $C = Sm \wedge \overline{Dm} \vee Rm \wedge \overline{Dm} \vee Sm \wedge Rm$ $Z = Z \wedge \overline{Rm} \wedge \ldots \wedge \overline{R0}$
CAS, CAS2, CMP, CMPI, CMPM	—	*	*	?	?	$V = \overline{Sm} \wedge Dm \wedge \overline{Rm} \vee Sm \wedge \overline{Dm} \wedge Rm$ $C = Sm \wedge \overline{Dm} \vee Rm \wedge \overline{Dm} \vee Sm \wedge Rm$
DIVS, DUVI	—	*	*	?	0	V = Division Overflow
MULS, MULU	—	*	*	?	0	V = Multiplication Overflow
SBCD, NBCD	*	U	?	U	?	C = Decimal Borrow $Z = Z \wedge \overline{Rm} \wedge \ldots \wedge \overline{Ro}$
NEG	*	*	*	?	?	$V = Dm \wedge Rm$ $C = Dm \vee Rm$
NEGX	*	*	?	?	?	$V = Dm \wedge Rm$ $C = Dm \vee Rm$ $Z = Z \wedge \overline{Rm} \wedge \ldots \wedge \overline{R0}$
BTST, BCHG, BSET, BCLR	—	—	?	—	—	$Z = \overline{Dn}$
BFTST, BFCHG, BFSET, BFCLR	—	?	?	0	0	$N = Dm$ $Z = \overline{Dm} \wedge \overline{DM-1} \wedge \ldots \wedge \overline{D0}$

Table 3-12. Condition Code Computations (Sheet 2 of 2)

Operations	X	N	Z	V	C	Special Definition
BFEXTS, BFEXTU, BFFFO	—	?	?	0	0	$N = Sm$ $Z = \overline{Sm} \backslash \overline{Sm - 1} \backslash \ldots \backslash \overline{S0}$
BFINS	—	?	?	0	0	$N = Dm$ $Z = \overline{Dm} \backslash \overline{DM - 1} \backslash \ldots \backslash \overline{D0}$
ASL	*	*	*	?	?	$V = Dm \backslash (\overline{Dm - 1} V \ldots V \overline{Dm - r}) V \overline{Dm} \backslash (DM - 1 V \ldots + Dm - r)$ $C = \overline{Dm - r + 1}$
ASL (R = 0)	—	*	*	0	0	
LSL, ROXL	*	*	*	0	?	$C = Dm - r + 1$
LSR (r = 0)	—	*	*	0	0	
ROXL (r = 0)	—	*	*	0	?	$C = X$
ROL	—	*	*	0	?	$C = Dm - r + 1$
ROL (r = 0)	—	*	*	0	0	
ASR, LSR, ROXR	*	*	*	0	?	$C = Dr - 1$
ASR, LSR (r = 0)	—	*	*	0	0	
ROXR (r = 0)	—	*	*	0	?	$C = X$
ROR	—	*	*	0	?	$C = Dr - 1$
ROR (r = 0)	—	*	*	0	0	

— = Not Affected
U = Undefined, Result Meaningless
? = Other — See Special Definition
* = General Case
  $X = C$
  $N = Rm$
  $Z = \overline{Rm} \backslash \ldots \backslash \overline{R0}$
Sm = Source Operand — Most Significant Bit
Dm = Destination Operand — Most Significant Bit

Rm = Result Operand — Most Significant Bit
R = Register Tested
n = Bit Number
r = Shift Count
LB = Lower bound
UB = Upper Bound
\V = Boolean AND
V = Boolean OR
$\overline{Rm}$ = NOT Rm

3.8.2 Condition Tests

Table 3-13 lists the condition names, encodings, and tests for the condition branch and set instructions. The test associated with each condition is a logical formula using the current states of the condition codes. If this formula evaluates to one, the condition is true. If the formula evaluates to zero, the condition is false. For example, the T condition is always true, and the EQ condition is true only if the Z bit condition code is currently true.

3.9 INSTRUCTION FORMAT SUMMARY

The following paragraphs provide a summary of the primary words in each instruction of the instruction set. The complete instruction definition consists of the primary words followed by the addressing mode operands such as immediate data fields, displacements, and index operands. The four most significant bits of the first (or only) primary word provide a means of categorizing the instructions. Table 3-14 is an operation code (opcode) map that lists an instruction category for each combination of these bits.

Table 3-13. Conditional Tests

Mnemonic	Condition	Encoding	Test
T*	True	0000	1
F*	False	0001	0
HI	High	0010	$\overline{C} \bullet \overline{Z}$
LS	Low or Same	0011	$C + Z$
CC(HS)	Carry Clear	0100	$\overline{C}$
CS(LO)	Carry Set	0101	C
NE	Not Equal	0110	$\overline{Z}$
EQ	Equal	0111	Z
VC	Overflow Clear	1000	$\overline{V}$
VS	Overflow Set	1001	V
PL	Plus	1010	$\overline{N}$
MI	Minus	1011	N
GE	Greater or Equal	1100	$N \bullet V + \overline{N} \bullet \overline{V}$
LT	Less Than	1101	$N \bullet \overline{V} + \overline{N} \bullet V$
GT	Greater Than	1110	$N \bullet V \bullet \overline{Z} + \overline{N} \bullet \overline{V} \bullet \overline{Z}$
LE	Less or Equal	1111	$Z + N \bullet \overline{V} + \overline{N} \bullet V$

$\bullet$ = Boolean AND
$+$ = Boolean OR
$\overline{N}$ = Boolean NOT N

*Not available for the Bcc instruction.

Table 3-14. Operation Code Map

Bits 15 through 12	Operation
0000	Bit Manipulation/MOVEP/Immediate
0001	Move Byte
0010	Move Long
0011	Move Word
0100	Miscellaneous
0101	ADDQ/SUBQ/Scc/DBcc/TRAPcc
0110	Bcc/BSR/BRA
0111	MOVEQ
1000	OR/DIV/SBCD
1001	SUB/SUBX
1010	(Unassigned, Reserved)
1011	CMP/EOR
1100	AND/MUL/ABCD/EXG
1101	ADD/ADDX
1110	Shift/Rotate/Bit Field
1111	Coprocessor Interface

The first portion of the instruction summary groups the standard instructions according to the opcode map. The next portion documents coprocessor instruction forms.

MC68020 INSTRUCTIONS

ORI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	SIZE		EFFECTIVE ADDRESS MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Size Field: 00 = Byte 01 = Word 10 = Long

ORI to CCR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

ORI to SR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
WORD DATA (16 BITS)															

CMP2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	SIZE		0	1	1	EFFECTIVE ADDRESS MODE			REGISTER		
D/A	REGISTER			0	0	0	0	0	0	0	0	0	0	0	0

Size Field: 00 = Byte 01 = Word 10 = Long

CHK2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	SIZE		0	1	1	EFFECTIVE ADDRESS MODE			REGISTER		
D/A	REGISTER			1	0	0	0	0	0	0	0	0	0	0	0

Size Field: 00 = Byte 01 = Word 10 = Long

Bit (Dynamic)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	DATA REGISTER			1	TYPE		EFFECTIVE ADDRESS MODE			REGISTER		

Type Field: 00 = TST 10 = CLR 01 = CHG 11 = SET

MOVEP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	DATA REGISTER			OPCODE			0	0	1	ADDRESS REGISTER		
DISPLACEMENT (16 BITS)															

Opmode Field: 100 = Transfer Word from Memory to Register
101 = Transfer Long from Memory to Register
110 = Transfer Word from Register to Memory
111 = Transfer Long from Register to Memory

ANDI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Size Field: 00 = Byte 01 = Word 10 = Long

ANDI to CCR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

ANDI to SR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0
WORD DATA (16 BITS)															

SUBI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Size Field: 00 = Byte 01 = Word 10 = Long

ADDI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Size Field: 00 = Byte 01 = Word 10 = Long

RTM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	0	0	D/A	REGISTER		

CALLM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	EFFECTIVE ADDRESS MODE			REGISTER		
0	0	0	0	0	0	0	0	ARGUMENT COUNT							

CAS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	SIZE		0	1	1	EFFECTIVE ADDRESS MODE			REGISTER		
0	0	0	0	0	0	0	Du			0	0	0	Dc		

Size Field: 01 = Byte 10 = Word 11 = Long

CAS2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	SIZE		0	1	1	1	1	1	1	0	0
D/A1	Rn1			0	0	0	Du1			0	0	0	Dc1		
D/A2	Rn2			0	0	0	Du2			0	0	0	Dc2		

Size Field: 10 = Word 11 = Long

Bit (Static)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	TYPE		EFFECTIVE ADDRESS MODE			REGISTER		
0	0	0	0	0	0	0	0	BIT NUMBER							

Type Field: 00 = TST 10 = CLR 01 = CHG 11 = SET

EORI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	SIZE		EFFECTIVE ADDRESS MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Size Field: 00 = Byte 01 = Word 10 = Long

EORI to CCR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

EORI to SR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0
WORD DATA (16 BITS)															

CMPI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Size Field: 00 = Byte 01 = Word 10 = Long

MOVES

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
A/D	REGISTER			dr	0	0	0	0	0	0	0	0	0	0	0

dr Field: 0 = EA to Register 1 = Register to EA

MOVE Byte

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	DESTINATION						SOURCE					
				REGISTER			MODE			MODE			REGISTER		

Note Register and Mode Locations

MOVEA Long

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	DESTINATION REGISTER			0	0	1	SOURCE					
										MODE			REGISTER		

MOVE Long

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	DESTINATION						SOURCE					
				REGISTER			MODE			MODE			REGISTER		

Note Register and Mode Locations

MOVEA Word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	DESTINATION REGISTER			0	0	1	SOURCE					
										MODE			REGISTER		

MOVE Word

				DESTINATION		SOURCE	
0	0	1	1	REGISTER	MODE	MODE	REGISTER

Note Register and Mode Locations

NEGX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Size Field: 00 = Byte 01 = Word 10 = Long

MOVE from SR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

CHK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	REGISTER			SIZE		0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Size Field: 10 = Long 11 = Word

LEA

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	REGISTER			1	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

CLR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Size Field: 00 = Byte 01 = Word 10 = Long

MOVE from CCR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

NEG

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Size Field: 00 = Byte 01 = Word 10 = Long

MOVE to CCR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

NOT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Size Field: 00 = Byte 01 = Word 10 = Long

MOVE to SR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

NBCD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

LINK Long

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	0	1	REGISTER		
HIGH-ORDER DISPLACEMENT															
LOW-ORDER DISPLACEMENT															

SWAP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	0	REGISTER		

BKPT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	1	VECTOR		

PEA

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Size Field: 00 = Byte 01 = Word 10 = Long

EXT/EXTB

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	OPCODE			0	0	0	REGISTER		

Opmode Field: 010 = Extend Word 011 = Extend Long 111 = Extend Byte Long

MOVEM Registers to EA

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	dr	0	0	1	SIZE	EFFECTIVE ADDRESS					
										MODE			REGISTER		
REGISTER LIST MASK															

Size Field: 0 = Word Transfer 1 = Long Transfer

TST

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Size Field: 00 = Byte 01 = Word 10 = Long

TAS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

ILLEGAL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	1	1	1	1	0	0

MULS/MULU Long

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dl			TYPE	SIZE	0	0	0	0	0	0	0	REGISTER Dh		

Type Field: 0 = MULU 1 = MULS
Size Field: 0 = Longword Product 1 = Quadword Product

DIVS/DIVU Long
DIVUL/DIVSL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dq			TYPE	SIZE	0	0	0	0	0	0	0	REGISTER Dr		

Type Field: 0 = DIVU 1 = DIVS
Size Field: 0 = Longword Dividend 1 = Quadword Dividend

MOVEM EA to Registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	dr	0	0	1	SIZE	EFFECTIVE ADDRESS					
										MODE			REGISTER		
REGISTER LIST MASK															

Size Field: 0 = Word Transfer 1 = Long Transfer

TRAP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	0	VECTOR			

Link Word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	REGISTER		
WORD DISPLACEMENT															

UNLK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	REGISTER		

MOVE to USP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	0	REGISTER		

MOVE from USP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	1	REGISTER		

RESET

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0

NOP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1

STOP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0

RTE

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1

RTD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	0
DISPLACEMENT (16 BITS)															

RTS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1

TRAPV

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	0

RTR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	1

MOVEC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	dr
A/D	REGISTER			CONTROL REGISTER											

dr Field: 0 = Control Register to General Register
1 = General Register to Control Register

Control Register Field:
$000 = SFC
$001 = DFC
$002 = CACR
$800 = USP
$801 = VBR
$802 = CAAR
$803 = MSP
$804 = ISP

JSR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

JMP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

ADDQ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	DATA			0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Data Field: Three bits of immediate data, 1-7 represent immediate values of 1-7, and 0 represents 8.
Size Field: 00 = Byte 01 = Word 10 = Long

Scc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	CONDITION				1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

DBcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	CONDITION				1	1	0	0	1	REGISTER		
DISPLACEMENT (16 BITS)															

TRAPcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	CONDITION				1	1	1	1	1	OPMODE		
OPTIONAL WORD															
OR LONG WORD															

Opmode Field: 010 = Word Operand 011 = Long Operand 100 = No Operand

SUBQ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	DATA			1	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Data Field: Three bits of immediate data; 1-7 represent immediate values of 1-7, and 0 represents 8.
Size Field: 00 = Byte 01 = Word 10 = Long

Bcc

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td colspan="4">CONDITION</td><td colspan="8">8-BIT DISPLACEMENT</td></tr>
<tr><td colspan="16">16-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $00</td></tr>
<tr><td colspan="16">32-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $FF</td></tr>
</table>

BRA

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="8">8-BIT DISPLACEMENT</td></tr>
<tr><td colspan="16">16-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $00</td></tr>
<tr><td colspan="16">32-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $FF</td></tr>
</table>

BSR

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="8">8-BIT DISPLACEMENT</td></tr>
<tr><td colspan="16">16-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $00</td></tr>
<tr><td colspan="16">32-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $FF</td></tr>
</table>

MOVEQ

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td colspan="3">REGISTER</td><td>0</td><td colspan="8">DATA</td></tr>
</table>

Data Field: Data is sign extended to a long operand and all 32 bits are transferred to the data register.

OR

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2" colspan="3">REGISTER</td><td rowspan="2" colspan="3">OPMODE</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Opmode Field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩)v(⟨Dn⟩) ➧ ⟨Dn⟩
100	101	110	(⟨Dn⟩)v(⟨ea⟩) ➧ ⟨ea⟩

DIVS/DIVU Word

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2" colspan="3">REGISTER</td><td rowspan="2">TYPE</td><td rowspan="2">1</td><td rowspan="2">1</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Type Field: 0 = DIVU 1 = DIVS

SBCD

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="3">REGISTER Ry</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>R/M</td><td colspan="3">REGISTER Rx</td></tr>
</table>

R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, Both Registers are Data Registers
If R/M = 1, Both Registers are Address Registers for the Predecrement Addressing Mode

PACK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	REGISTER Dy/Ay			1	0	1	0	0	R/M	REGISTER Dx/Ax		
16-BIT EXTENSION: ADJUSTMENT															

R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, Both Registers are Data Registers
If R/M = 1, Both Registers are Address Registers for the Predecrement Addressing Mode

UNPK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	REGISTER Dy/Ay			1	1	0	0	0	R/M	REGISTER Dx/Ax		
16-BIT EXTENSION: ADJUSTMENT															

R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, Both Registers are Data Registers
If R/M = 1, Both Registers are Address Registers for the Predecrement Addressing Mode

SUB

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Opmode Field:

Byte	Word	Long	Operation
000	001	010	(⟨Dn⟩) − (⟨ea⟩) ➧ ⟨Dn⟩
100	101	110	(⟨ea⟩) − (⟨Dn⟩) ➧ ⟨ea⟩

SUBA

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Opmode Field:

Word	Long	Operation
011	111	(⟨An⟩) − (⟨ea⟩) ➧ ⟨An⟩

SUBX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	REGISTER Ry			1	SIZE		0	0	R/M	REGISTER Rx		

Size Field: 00 = Byte 01 = Word 10 = Long
R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, Both Registers are Data Registers
If R/M = 1, Both Registers are Address Registers for the Predecrement Addressing Mode

CMP

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td colspan="3" rowspan="2">REGISTER</td><td colspan="3" rowspan="2">OPMODE</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Opmode Field:

Byte	Word	Long	Operation
000	001	010	(⟨Dn⟩) – (⟨ea⟩)

CMPA

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td colspan="3" rowspan="2">REGISTER</td><td colspan="3" rowspan="2">OPMODE</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Opmode Field:

Word	Long	Operation
011	111	(⟨An⟩) – (⟨ea⟩)

EOR

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td colspan="3" rowspan="2">REGISTER</td><td colspan="3" rowspan="2">OPMODE</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Opmode Field:

Byte	Word	Long	Operation
100	101	110	(⟨ea⟩) ⊕ (⟨Dn⟩) → ⟨ea⟩

CMPM

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td colspan="3">REGISTER Ax</td><td>1</td><td colspan="2">SIZE</td><td>0</td><td>0</td><td>1</td><td colspan="3">REGISTER Ay</td></tr>
</table>

Size Field: 00 = Byte 01 = Word 10 = Long

AND

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="3" rowspan="2">REGISTER</td><td colspan="3" rowspan="2">OPMODE</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Opmode Field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩) \(⟨Dn⟩) → ⟨Dn⟩
100	101	110	(⟨Dn⟩) \(⟨ea⟩) → ⟨ea⟩

MULS/MULU Word

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="3" rowspan="2">REGISTER</td><td rowspan="2">TYPE</td><td rowspan="2">1</td><td rowspan="2">1</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Type Field: 0 = MULU 1 = MULS

ABCD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER Rx			1	0	0	0	0	R/M	REGISTER Ry		

R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, Both Registers are Data Registers
If R/M = 1, Both Registers are Address Registers for the Predecrement Addressing Mode

EXG Data Registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER Dx			1	0	1	0	0	0	REGISTER Dy		

EXG Address Registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER Ax			1	0	1	0	0	1	REGISTER Ay		

EXG Data Register and Address Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER Dx			1	1	0	0	0	1	REGISTER Ay		

ADD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Opmode Field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩) + (⟨Dn⟩) → ⟨Dn⟩
100	101	110	(⟨Dn⟩) + (⟨ea⟩) → ⟨ea⟩

ADDA

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Opmode Field:

Word	Long	Operation
011	111	(⟨ea⟩) + (⟨An⟩) → ⟨An⟩

ADDX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	REGISTER Rx			1	SIZE		0	0	R/M	REGISTER Ry		

Size Field: 00 = Byte 01 = Word 10 = Long
R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, Both Registers are Data Registers
If R/M = 1, Both Registers are Address Registers for the Predecrement Addressing Mode

Shift/Rotate Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	COUNT/REGISTER			dr	SIZE		I/R	TYPE		REGISTER		

Count/Register Field:
If I/R Field = 0, Specifies Shift Count
If I/R Field = 1, Specifies a Data Register that Contains the Shift Count
dr Field: 0 = Right 1 = Left
Size Field: 00 = Byte 01 = Word 10 = Long
I/R Field: 0 = Immediate Shift Count 1 = Register Shift Count
Type Field: 00 = Arithmetic Shift 01 = Logical Shift 10 = Rotate with Extend 11 = Rotate

Shift/Rotate Memory

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										EFFECTIVE ADDRESS					
1	1	1	0	0	TYPE		dr	1	1	MODE			REGISTER		

Type Field: 00 = Arithmetic Shift 01 = Logical Shift 10 = Rotate with Extend 11 = Rotate
dr Field: 0 = Right 1 = Left

Bit Field

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										EFFECTIVE ADDRESS					
1	1	1	0	1	TYPE			1	1	MODE			REGISTER		
0	REGISTER			Do	OFFSET					Dw	WIDTH				

Type Field: 000 = BFTST 100 = BFCLR
001 = BFEXTU 101 = BFFFO
010 = BFCHG 110 = BFSET
011 = BFEXTS 111 = BFINS

Register Field is 000 for BFTST, BFCHG, BFCLR, and BFSET, Otherwise, Register Field is Destination Register
Do Field: 0 = Offset is Immediate 1 = Offset is in Data Register
Dw Field: 0—Width is Immediate 1 = Width is in Data Register

COPROCESSOR INSTRUCTIONS

cpGEN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
COPROCESSOR-DEPENDENT COMMAND WORD															
OPTIONAL EFFECTIVE ADDRESS OR COPROCESSOR-DEFINED EXTENSION WORDS															

cpScc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	0	0	0	0	0	0	0	0	0	COPROCESSOR CONDITION					
OPTIONAL EFFECTIVE ADDRESS OR COPROCESSOR-DEFINED EXTENSION WORDS															

cpDBcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			0	0	1	0	0	1	REGISTER		
0	0	0	0	0	0	0	0	0	0	COPROCESSOR CONDITION					
OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS															
DISPLACEMENT (16 BIT)															

cpTRAPcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			0	0	1	1	1	1	MODE		
0	0	0	0	0	0	0	0	0	0	COPROCESSOR CONDITION					
OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS															
OPTIONAL WORD															
OR LONG WORD OPERAND															

Mode Field: 010 = Word Operand 011 = Longword Operand 100 = No Displacement

cpBcc

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			0	1	SIZE	COPROCESSOR CONDITION					
OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS															
WORD OR															
LONG WORD DISPLACEMENT															

Size Field: 0 = Word Displacement 1 = Longword Displacement

cpSAVE

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			1	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

cpRESTORE

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CP-ID			1	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

SECTION 4
PROCESSING STATES

This section describes the processing states of the MC68020. It describes the functions of the bits in the supervisor portion of the status register, and the actions taken by the processor in response to exception conditions.

Unless the processor has halted, it is always in either the normal or the exception processing state. Whenever the processor is executing instructions or fetching instructions or operands it is in the normal processing state. The processor is also in the normal processing state while it is storing instruction results or communicating with a coprocessor.

NOTE

> Exception processing refers specifically to the transition from normal processing of a program to normal processing of system routines, interrupt routines, and other exception handlers. Exception processing includes all stacking operations, the fetch of the exception vector, and filling of the instruction pipe caused by an exception. It has completed when execution of the first instruction of the exception handler routine begins.

The processor enters the exception processing state when an interrupt is acknowledged, when an instruction is traced or results in a trap, or when some other exceptional condition arises. Execution of certain instructions or unusual conditions occurring during the execution of any instructions can cause exceptions. External conditions, such as interrupts, bus errors, and some coprocessor responses also cause exceptions. Exception processing provides an efficient transfer of control to handlers and routines that process the exceptions.

A catastrophic system failure occurs whenever the processor receives a bus error or generates an address error while in the exception processing state. This type of failure halts the processor. For example, if during the exception processing of one bus error another bus error occurs, the MC68020 has not completed the transition to normal processing and has not completed saving the internal state of the machine, so the processor assumes that the system is not operational and halts. Only an external reset can restart a halted processor. (When the processor executes a STOP instruction, it is in a special type of normal processing state, one without bus cycles. It is stopped, not halted.)

4.1 PRIVILEGE LEVELS

The processor operates at one of two levels of privilege: the user level or the supervisor level. The supervisor level has higher privileges than the user level. Not all processor or coprocessor instructions are permitted to execute in the lower-privileged user level, but all are available at the supervisor level. This allows a separation of supervisor and user so the supervisor can protect system resources from uncontrolled access. The processor uses the privilege level indicated by the S bit in the status register to select either the user or supervisor privilege level and either the user stack pointer or a supervisor stack pointer for stack operations. The processor identifies a bus access (supervisor or user mode) via the function codes so that differentiation between supervisor and user can be maintained.

In many systems, the majority of programs execute at the user level. User programs can access only their own code and data areas and can be restricted from accessing other information. The operating system typically executes at the supervisor privilege level. It has access to all resources, performs the overhead tasks for the user level programs, and coordinates their activities.

4.1.1 Supervisor Privilege Level

The supervisor level is the higher privilege level. The privilege level is determined by the S bit of the status register; if the S bit is set, the supervisor privilege level applies, and all instructions are executable. The bus cycles for instructions executed at the supervisor level are normally classified as supervisor references, and the values of the function codes on FC2–FC0 refer to supervisor address spaces.

In a multitasking operating system, it is more efficient to have a supervisor stack space associated with each user task and a separate stack space for interrupt associated tasks. The MC68020 provides two supervisor stacks, master and interrupt; the M bit of the status register selects which of the two is active. When the M bit is set to one, supervisor stack pointer references (either implicit or by specifying address register A7) access the master stack pointer, MSP. The operating system sets the MSP for each task to point to a task-related area of supervisor data space. This separates task-related supervisor activity from asynchronous, I/O-related supervisor tasks that may be only coincidental to the currently executing task. The master stack (MSP) can separately maintain task control information for each currently executing user task, and the software updates the MSP pointer when a task switch is performed, providing an efficient means for transferring task related stack items. The other supervisor stack (ISP) can be used for interrupt control information and workspace area as interrupt handling routines require.

When the M bit is clear, the MC68020 is in the interrupt mode of the supervisor privilege level, and operation is the same as in the MC68000, MC68008, and MC68010 supervisor mode. (The processor is in this mode after a reset operation.) All supervisor stack pointer references access the interrupt stack pointer (ISP) in this mode.

The value of the M bit in the status register does not affect execution of privileged instructions; both master and interrupt modes are at the supervisor privilege level. Instructions that affect the M bit are MOVE to SR, ANDI to SR, EORI to SR, ORI to SR, and RTE. Also, the processor automatically saves the M bit value and clears it in the status register as part of the exception processing for interrupts.

All exception processing is performed at the supervisor privilege level. All bus cycles generated during exception processing are supervisor references, and all stack accesses use the active supervisor stack pointer.

4.1.2 User Privilege Level

The user level is the lower-privilege level. The privilege level is determined by the S bit of the status register; if the S bit is clear, the processor executes instructions at the user privilege level.

Most instructions execute at either privilege level, but some instructions that have important system effects are privileged and can only be executed at the supervisor level. For instance, user programs are not allowed to execute the STOP instruction or the RESET instruction. To prevent a user program from entering the supervisor privilege level, except in a controlled manner, instructions that can alter the S bit in the status register are privileged. The TRAP #n instruction provides controlled access to operating system services for user programs.

The bus cycles for an instruction executed at the user privilege level are classified as user references, and the values of the function codes on FC2–FC0 specify user address spaces. While the processor is at the user level, references to the system stack pointer implicitly, or to address register seven (A7) explicitly, refer to the user stack pointer (USP).

4.1.3 Changing Privilege Level

In order to change from the user to the supervisor privilege level, one of the conditions that causes the processor to perform exception processing must occur. This causes a change from the user level to the supervisor level and can cause a change from the master mode to the interrupt mode. Exception processing saves the current values of the S and M bits of the status register (along with the rest of the status register) on the active supervisor stack, and then sets the S bit, forcing the processor into the supervisor privilege level. When the exception being processed is an interrupt and the M bit is set, the M bit is cleared putting the processor into the interrupt mode. Execution of instructions continues at the supervisor level to process the exception condition.

To return to the user privilege level, a system routine must execute one of the following instructions: MOVE to SR, ANDI to SR, EORI to SR, ORI to SR, or RTE. The MOVE, ANDI, EORI, and ORI to SR and RTE instructions execute at the supervisor privilege level, and can modify the S bit of the status register. After these instructions execute, the instruction pipeline is flushed and is refilled from the appropriate address space.

The RTE instruction returns to the program that was executing when the exception occurred. It restores the exception stack frame saved on the supervisor stack. If the frame on top of the stack was generated by an interrupt, trap, or instruction exception, the RTE instruction restores the status register and program counter to the values saved on the supervisor stack. The processor then continues execution at the restored program counter address and at the privilege level determined by the S bit of the restored status register. If the frame on top of the stack was generated by a bus fault (bus error or address error exception), the RTE instruction restores the entire saved processor state from the stack.

4.2 ADDRESS SPACE TYPES

The processor specifies a target address space for every bus cycle with the function code signals according to the type of access required. In addition to distinguishing between supervisor/user and program/data, the processor can identify special processor cycles, such as the interrupt acknowledge cycle, and the memory management unit can control accesses and translate addresses appropriately. Table 4-1 lists the types of accesses defined for the MC68020 and the corresponding values of function codes FC2–FC0.

Table 4-1. Address Space Encodings

FC2	FC1	FC0	Address Space
0	0	0	(Undefined Reserved)*
0	0	1	User Data Space
0	1	0	User Program Space
0	1	1	(Undefined, Reserved)*
1	0	0	(Undefined, Reserved)*
1	0	1	Supervisor Data Space
1	1	0	Supervisor Program Space
1	1	1	CPU Space

*Address space 3 is reserved for user definition, while 0 and 4 are reserved for future use by Motorola.

The memory locations of user program and data accesses are not predefined. Neither are the locations of supervisor data space. During reset, the first two long words beginning at memory location zero in the supervisor program space are used for processor initialization. No other memory locations are explicitly defined by the MC68020.

A function code of $7 ([FC2:FC0] = 111) selects the CPU address space. This is a special address space that does not contain instructions or operands, but is reserved for special processor functions. The processor uses accesses in this space to communicate with external devices for special purposes. For example, all M68000 processors use the CPU space for interrupt acknowledge cycles. The MC68020 also generates CPU space accesses for breakpoint acknowledge and coprocessor operations.

Supervisor programs can use the MOVES instruction to access all address spaces, including the user spaces and the CPU address space. Although the MOVES instruction can be used to generate CPU space cycles, this may interfere with proper system operation. Thus, the use of MOVES to access the CPU space should be done with caution.

4.3 EXCEPTION PROCESSING

An exception is defined as a special condition that pre-empts normal processing. Both internal and external conditions cause exceptions. External conditions that cause exceptions are interrupts from external devices, bus errors, coprocessor detected errors, and reset. Instructions, address errors, tracing, and breakpoints are internal conditions that cause exceptions. The TRAP, TRAPcc, TRAPV, cpTRAPcc, CHK, CHK2, RTE, and DIV instructions can all generate exceptions as part of their normal execution. In addition, illegal instructions, privilege violations, and coprocessor protocol violations cause exceptions.

Exception processing, which is the transition from the normal processing of a program to the processing required for the exception condition, involves the exception vector table and an exception stack frame. The following paragraphs describe the vector table and a generalized exception stack frame. Exception processing is discussed in detail in **SECTION 8 EXCEPTION PROCESSING**. Coprocessor detected exceptions are discussed in detail in **SECTION 9 COPROCESSOR INTERFACE DESCRIPTION**.

4.3.1 Exception Vectors

The vector base register contains the base address of the 1024-byte exception vector table, which consists of 256 exception vectors. Exception vectors contain the memory addresses of routines that begin execution at the completion of exception processing. These routines perform a series of operations appropriate for the corresponding exceptions. Because the exception vectors contain memory addresses, each consists of one long word, except for the reset vector. The reset vector consists of two long words: the address used to initialize the interrupt stack pointer and the address used to initialize the program counter.

The address of an exception vector is derived from an 8-bit vector number and the vector base register (VBR). The vector numbers for some exceptions are obtained from an external device; others are supplied automatically by the processor. The processor multiplies the vector number by four to calculate the vector offset, which it adds to the VBR. The sum is the memory address of the vector. All exception vectors are located in supervisor data space, except the reset vector, which is located in supervisor program space. Only the initial reset vector is fixed in the processor's memory map; once initialization is complete, there are no fixed assignments. Since the VBR provides the base address of the vector table, the vector table can be located anywhere in memory; it can even be dynamically relocated for each task that is executed by an operating system. Details of exception processing are provided in **SECTION 8 EXCEPTION PROCESSING**, and Table 8-1 lists the exception vector assignments.

4.3.2 Exception Stack Frame

Exception processing saves the most volatile portion of the current processor context on the top of the supervisor stack. This context is organized in a format called the exception stack frame. This information always includes a copy of the status register, the program counter, the vector offset of the vector, and the frame format field. The frame format field identifies the type of stack frame. The RTE instruction uses the value in the format field to properly restore the information stored in the stack frame and to deallocate the stack space. The general form of the exception stack frame is illustrated in Figure 4-1. Refer to **SECTION 8 EXCEPTION PROCESSING** for a complete list of exception stack frames.

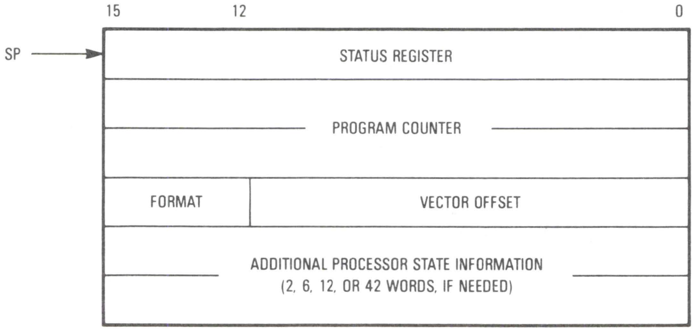

15
12
0
SP
STATUS REGISTER
PROGRAM COUNTER
FORMAT
VECTOR OFFSET
ADDITIONAL PROCESSOR STATE INFORMATION
(2, 6, 12, OR 42 WORDS, IF NEEDED)

Figure 4-1. General Exception Stack Frame

SECTION 5
SIGNAL DESCRIPTION

This section contains brief descriptions of the input and output signals in their functional groups, as shown in Figure 5-1. Each signal is explained in a brief paragraph with reference to other sections that contain more detail about the signal and the related operations.

NOTE

In this section and in the remainder of the manual, **assertion** and **negation** are used to specify forcing a signal to a particular state. In particular, assertion and assert refer to a signal that is active or true; negation and negate indicate a signal that is inactive or false. These terms are used independently of the voltage level (high or low) that they represent.

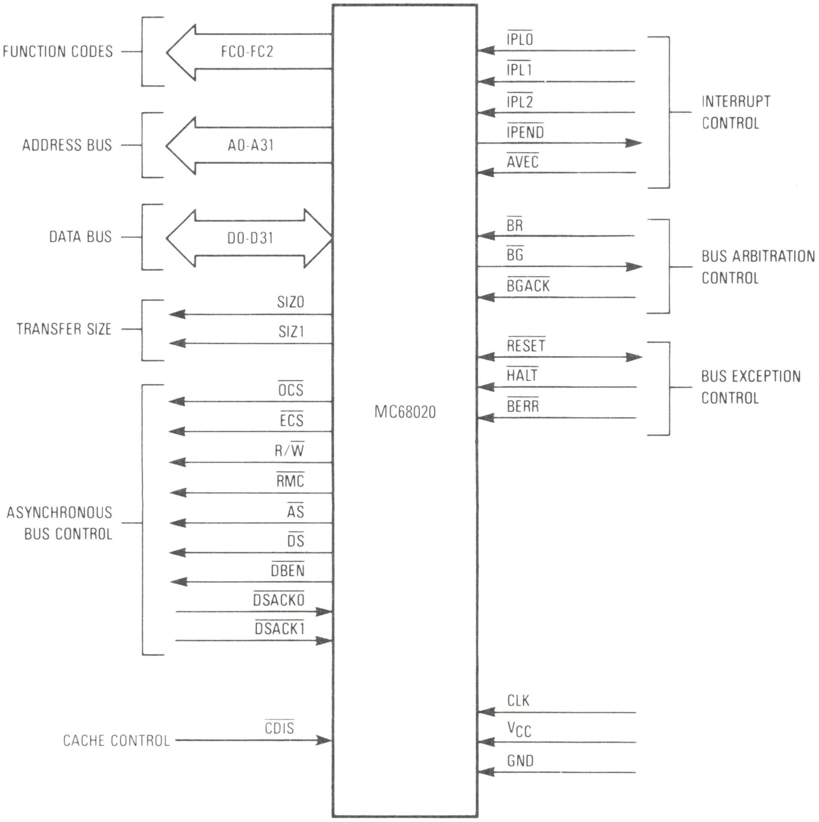

FUNCTION CODES
FC0-FC2
ADDRESS BUS
A0-A31
DATA BUS
D0-D31
TRANSFER SIZE
SIZ0
SIZ1
ASYNCHRONOUS BUS CONTROL
OCS
ECS
R/W
RMC
AS
DS
DBEN
DSACK0
DSACK1
CACHE CONTROL
CDIS
MC68020
IPL0
IPL1
IPL2
IPEND
AVEC
INTERRUPT CONTROL
BR
BG
BGACK
BUS ARBITRATION CONTROL
RESET
HALT
BERR
BUS EXCEPTION CONTROL
CLK
V_{CC}
GND

Figure 5-1. Functional Signal Groups

5.1 SIGNAL INDEX

The input and output signals for the MC68020 are listed in Table 5-1. Both the names and mnemonics are shown, along with brief descriptions of the signals. For more detail on each signal, refer to the paragraph in this section named for the signal, and the reference in that paragraph to a description of the related operations.

Guaranteed timing specifications, for the signals listed in Table 5-1, can be found in **SECTION 12 ELECTRICAL SPECIFICATIONS**.

Table 5-1. Signal Index

Signal Name	Mnemonic	Function
Function Codes	FC2–FC0	3-bit function code used to identify the address space of each bus cycle.
Address Bus	A31–A0	32-bit address bus.
Data Bus	D31–D0	32-bit data bus used to transfer 8, 16, 24, or 32 bits of data per bus cycle.
Size	SIZ1/SIZ0	Indicates the number of bytes remaining to be transferred for this cycle. These signals, together with A1 and A0, define the active sections of the data bus.
External Cycle Start	$\overline{\text{ECS}}$	Provides an indication that a bus cycle is beginning.
Operand Cycle Start	$\overline{\text{OCS}}$	Identical operation to that of $\overline{\text{ECS}}$ except that $\overline{\text{OCS}}$ is asserted only during the first bus cycle of an operand transfer.
Read/Write	R/$\overline{\text{W}}$	Defines the bus transfer as a processor read or write.
Read-Modify-Write Cycle	$\overline{\text{RMC}}$	Provides an indicator that the current bus cycle is part of an indivisible read-modify-write operation.
Address Strobe	$\overline{\text{AS}}$	Indicates that a valid address is on the bus.
Data Strobe	$\overline{\text{DS}}$	Indicates that valid data is to be placed on the data bus by an external device or has been placed on the data bus by the MC68020.
Data Buffer Enable	$\overline{\text{DBEN}}$	Provides an enable signal for external data buffers.
Data Transfer and Size Acknowledge	$\overline{\text{DSACK1}}$/ $\overline{\text{DSACK0}}$	Bus response signals that indicate the requested data transfer operation is completed. In addition, these two lines indicate the size of the external bus port on a cycle-by-cycle basis and are used for asynchronous transfers.
Interrupt Priority Level	$\overline{\text{IPL2}}$–$\overline{\text{IPL0}}$	Provides an encoded interrupt level to the processor.
Interrupt Pending	$\overline{\text{IPEND}}$	Indicates that an interrupt is pending.
Autovector	$\overline{\text{AVEC}}$	Requests an autovector during an interrupt acknowledge cycle.
Bus Request	$\overline{\text{BR}}$	Indicates that an external device requires bus mastership.
Bus Grant	$\overline{\text{BG}}$	Indicates that an external device may assume bus mastership.
Bus Grant Acknowledge	$\overline{\text{BGACK}}$	Indicates that an external device has assumed bus mastership.
Reset	$\overline{\text{RESET}}$	System reset.
Halt	$\overline{\text{HALT}}$	Indicates that the processor should suspend bus activity.
Bus Error	$\overline{\text{BERR}}$	Indicates that an erroneous bus operation is being attempted.
Cache Disable	$\overline{\text{CDIS}}$	Dynamically disables the on-chip cache to assist emulator support.
Clock	CLK	Clock input to the processor.
Power Supply	V_{CC}	Power supply.
Ground	GND	Ground connection.

5.2 FUNCTION CODE SIGNALS (FC2–FC0)

These three-state outputs identify the address space of the current bus cycle. Table 4-1 shows the relationships of the function code signals to the privilege levels and the address spaces. Refer to **4.2 ADDRESS SPACE TYPES** for more information.

5.3 ADDRESS BUS (A31–A0)

These three-state outputs provide the address for the current bus cycle, except in the CPU address space. Refer to **4.2 ADDRESS SPACE TYPES** for more information on the CPU address space. A31 is the most significant address signal. Refer to **7.1.2 Address Bus** for information on the address bus and its relationship to bus operation.

5.4 DATA BUS (D31–D0)

These three-state bidirectional signals provide the general-purpose data path between the MC68020 and all other devices. The data bus can transfer 8, 16, 24, or 32 bits of data per bus cycle. D31 is the most significant bit of the data bus. Refer to **7.1.4 Data Bus** for more information on the data bus and its relationship to bus operation.

5.5 TRANSFER SIZE SIGNALS (SIZ1, SIZ0)

These three-state outputs indicate the number of bytes remaining to be transferred for the current bus cycle. With A1, A0, $\overline{\mathrm{DSACK1}}$, and $\overline{\mathrm{DSACK0}}$, SIZ1 and SIZ0 define the number of bits transferred on the data bus. Refer to **7.2.1 Dynamic Bus Sizing** for more information on the size signals and their use in dynamic bus sizing.

5.6 BUS CONTROL SIGNALS

The following signals control synchronous bus transfer operations for the MC68020.

5.6.1 Operand Cycle Start ($\overline{\mathrm{OCS}}$)

This output signal indicates the beginning of the first external bus cycle for an instruction prefetch or a data operand transfer. $\overline{\mathrm{OCS}}$ is not asserted for subsequent cycles that are performed due to dynamic bus sizing or operand misalignment. Refer to **7.1.1 Bus Control Signals** for information about the relationship of $\overline{\mathrm{OCS}}$ to bus operation.

5.6.2 External Cycle Start ($\overline{\mathrm{ECS}}$)

This output signal indicates the beginning of a bus cycle of any type. Refer to **7.1.1 Bus Control Signals** for information about the relationship of $\overline{\mathrm{ECS}}$ to bus operation.

5.6.3 Read/Write (R/$\overline{\mathrm{W}}$)

This three-state output signal defines the type of bus cycle. A high level indicates a read cycle; a low level indicates a write cycle. Refer to **7.1.1 Bus Control Signals** for information about the relationship of R/$\overline{\mathrm{W}}$ to bus operation.

5.6.4 Read-Modify-Write Cycle ($\overline{\mathrm{RMC}}$)

This three-state output signal identifies the current bus cycle as part of an indivisible read-modify-write operation; it remains asserted during all bus cycles of the read-modify-write operation. Refer to **7.1.1 Bus Control Signals** for information about the relationship of $\overline{\mathrm{RMC}}$ to bus operation.

5.6.5 Address Strobe ($\overline{\mathrm{AS}}$)

This three-state output indicates that a valid address is on the address bus. The function code, size, and read/write signals are also valid when $\overline{\mathrm{AS}}$ is asserted. Refer to **7.1.3 Address Strobe** for information about the relationship of $\overline{\mathrm{AS}}$ to bus operation.

5.6.6 Data Strobe ($\overline{\mathrm{DS}}$)

During a read cycle, this three-state output indicates that an external device should place valid data on the data bus. During a write cycle, the data strobe indicates that the MC68020 has placed valid data on the bus. During two-clock synchronous write cycles, the MC68020 does not assert $\overline{\mathrm{DS}}$. Refer to **7.1.5 Data Strobe** for more information about the relationship of $\overline{\mathrm{DS}}$ to bus operation.

5.6.7 Data Buffer Enable ($\overline{\mathrm{DBEN}}$)

This output is an enable signal for external data buffers. This signal may not be required in all systems. The timing of this signal may preclude its use in a system that supports two-clock synchronous bus cycles. Refer to **7.1.6 Data Buffer Enable** for more information about the relationship of $\overline{\mathrm{DBEN}}$ to bus operation.

5.6.8 Data Transfer and Size Acknowledge ($\overline{\mathrm{DSACK1}}$, $\overline{\mathrm{DSACK0}}$)

These inputs indicate the completion of a requested data transfer operation. In addition, they indicate the size of the external bus port at the completion of each cycle. These signals apply only to asynchronous bus cycles. Refer to **7.1.7 Bus Cycle Termination Signals** for more information on these signals and their relationship to dynamic bus sizing.

5.7 INTERRUPT CONTROL SIGNALS

The following signals are the interrupt control signals for the MC68020.

5.7.1 Interrupt Priority Level Signals

These input signals provide an indication of an interrupt condition and the encoding of the interrupt level from a peripheral or external prioritizing circuitry. IPL2 is the most significant bit of the level number. For example, since the IPLn signals are active low, IPL2–IPL0 equal to $5 corresponds to an interrupt request at interrupt level 2. Refer to **8.1.9 Interrupt Exceptions** for information on MC68020 interrupts.

5.7.2 Interrupt Pending (IPEND)

This output signal indicates that an interrupt request has been recognized internally and exceeds the current interrupt priority mask in the status register (SR). This output is for use by external devices (coprocessors and other bus masters, for example) to predict processor operation on the following instruction boundaries. Refer to **8.1.9 Interrupts Exceptions** for interrupt information. Also, refer to **7.4.1 Interrupt Acknowledge Bus Cycles** for bus information related to interrupts.

5.7.3 Autovector (AVEC)

This input signal indicates that the MC68020 should generate an automatic vector during an interrupt acknowledge cycle. Refer to **7.4.1.2 AUTOVECTOR INTERRUPT ACKNOWLEDGE CYCLE** for more information about automatic vectors.

5.8 BUS ARBITRATION CONTROL SIGNALS

The following signals are the three bus arbitration control signals used to determine which device in a system is the bus master.

5.8.1 Bus Request (BR)

This input signal indicates that an external device needs to become the bus master. This is typically a ''wire-ORed'' input (but does not need to be constructed from open-collector devices). Refer to **7.7 BUS ARBITRATION** for more information.

5.8.2 Bus Grant (BG)

This output indicates that the MC68020 will release ownership of the bus master when the current processor bus cycle completes. Refer to **7.7.2 Bus Grant** for more information.

5.8.3 Bus Grant Acknowledge ($\overline{\text{BGACK}}$)

This input indicates that an external device has become the bus master. Refer to **7.7.3 Bus Grant Acknowledge** for more information.

5.9 BUS EXCEPTION CONTROL SIGNALS

The following signals are the bus exception control signals for the MC68020.

5.9.1 Reset ($\overline{\text{RESET}}$)

This bidirectional open-drain signal is used to initiate a system reset. An external reset signal resets the MC68020 as well as all external devices. A reset signal from the processor (asserted as part of the RESET instruction) resets external devices only; the internal state of the processor is not altered. Refer to **7.8 RESET OPERATION** for a description of reset bus operation and **8.1.1 Reset Exception** for information about the reset exception.

5.9.2 Halt ($\overline{\text{HALT}}$)

The halt signal indicates that the processor should suspend bus activity or, when used with $\overline{\text{BERR}}$, that the processor should retry the current cycle. Refer to **7.5 BUS EXCEPTION CONTROL CYCLES** for a description of the effects of $\overline{\text{HALT}}$ on bus operations.

5.9.3 Bus Error ($\overline{\text{BERR}}$)

The bus error signal indicates that an invalid bus operation is being attempted or, when used with $\overline{\text{HALT}}$, that the processor should retry the current cycle. Refer to **7.5 BUS EXCEPTION CONTROL CYCLES** for a description of the effects of $\overline{\text{BERR}}$ on bus operations.

5.10 EMULATOR SUPPORT SIGNALS

The following signal supports emulation by providing a means for an emulator to disable the on-chip cache by supplying internal status information to an emulator. Refer to **SECTION 11 APPLICATIONS INFORMATION** for more detailed information on emulation support.

5.10.1 Cache Disable ($\overline{\text{CDIS}}$)

The cache disable signal dynamically disables the on-chip cache to assist emulator support. Refer to **6.1 ON-CHIP CACHE ORGANIZATION AND OPERATION** for information about the cache; refer to **SECTION 11 APPLICATIONS INFORMATION** for a description of the use of this signal by an emulator. $\overline{\text{CDIS}}$ does not flush the instruction cache; entries remain unaltered, and become available again when $\overline{\text{CDIS}}$ is negated.

5.11 CLOCK (CLK)

The clock signal is the clock input to the MC68020. It is a TTL-compatible signal. Refer to **SECTION 11 APPLICATIONS INFORMATION** for suggestions on clock generation and to **SECTION 12 ELECTRICAL SPECIFICATIONS** for electrical specifications.

5.12 POWER SUPPLY CONNECTIONS

The MC68020 requires connection to a V_{CC} power supply, positive with respect to ground. The V_{CC} connections are grouped to supply adequate current for the various sections of the processor. The ground connections are similarly grouped. **SECTION 13 ORDERING INFORMATION AND MECHANICAL DATA** describes the groupings of V_{CC} and ground connections, and **SECTION 11 APPLICATIONS INFORMATION** describes a typical power supply interface.

5.13 SIGNAL SUMMARY

Table 5-2 provides a summary of the electrical characteristics of the signals discussed in this section.

Table 5-2. Signal Summary

Signal Function	Signal Name	Input/Output	Active State	Three-State
Function Codes	FC2–FC0	Output	High	Yes
Address Bus	A31–A0	Output	High	Yes
Data Bus	D31–D0	Input Output	High	Yes
Transfer Size	SIZ1, SIZ0	Output	High	Yes
Operand Cycle Start	$\overline{\text{OCS}}$	Output	Low	No
External Cycle Start	$\overline{\text{ECS}}$	Output	Low	No
Read Write	R $\overline{\text{W}}$	Output	High Low	Yes
Read-Modify-Write Cycle	$\overline{\text{RMC}}$	Output	Low	Yes
Address Strobe	$\overline{\text{AS}}$	Output	Low	Yes
Data Strobe	$\overline{\text{DS}}$	Output	Low	Yes
Data Buffer Enable	$\overline{\text{DBEN}}$	Output	Low	Yes
Data Transfer and Size Acknowledge	$\overline{\text{DSACK1}}$, $\overline{\text{DSACK0}}$	Input	Low	—
Interrupt Priority Level	$\overline{\text{IPL2}}$–$\overline{\text{IPL0}}$	Input	Low	—
Interrupt Pending	$\overline{\text{IPEND}}$	Output	Low	No
Autovector	$\overline{\text{AVEC}}$	Input	Low	—
Bus Request	$\overline{\text{BR}}$	Input	Low	—
Bus Grant	$\overline{\text{BG}}$	Output	Low	No
Bus Grant Acknowledge	$\overline{\text{BGACK}}$	Input	Low	—
Reset	$\overline{\text{RESET}}$	Input Output	Low	No
Halt	$\overline{\text{HALT}}$	Input	Low	—
Bus Error	$\overline{\text{BERR}}$	Input	Low	—
Cache Disable	$\overline{\text{CDIS}}$	Input	Low	—
Clock	CLK	Input	—	—
Power Supply	V_{CC}	Input	—	—
Ground	GND	Input	—	—

SECTION 6
ON-CHIP CACHE MEMORY

The MC68020 incorporates an on-chip cache memory as a means of improving the performance of the processor. The cache is implemented as a CPU instruction cache and is used to store the instruction stream prefetch accesses from the main memory.

Reduced external bus activity increases overall performance by increasing the availability of the bus for use by external devices (in systems with more than one bus master, such as a processor and a DMA device) without degrading the performance of the MC68020. An increase in instruction throughput results when instruction words required by a program are available in the on-chip cache, and the time required to access them on the external bus is eliminated.

6.1 ON-CHIP CACHE ORGANIZATION AND OPERATION

The MC68020 on-chip instruction cache is a direct-mapped cache of 64 long word entries. Each cache entry consists of a tag field made up of the upper 24 address bits and the FC2 value, one valid bit and 32 bits (two words) of instruction data.

Figure 6-1 shows a block diagram of the on-chip cache.

When enabled, the instruction cache is used to store instruction prefetches (instruction words and extension words) as they are requested by the CPU. Instruction prefetches are normally requested from sequential memory addresses except when a change of program flow occurs (e.g., a branch taken) or when an instruction is executed that can modify the status register, in which cases the instruction pipe is automatically flushed and refilled.

Whenever an instruction fetch occurs, the cache (if enabled) is first checked to determine if the word required is in the cache. This is achieved by first using the index field (A7–A4) of the access address as an index into the on-chip cache. This selects one of the 64 entries in the cache. Next, the access address bits A31–A8, and FC2 are compared to the tag of the selected entry. If there is a match and the valid bit is set, a cache hit occurs. Address bit A1 is used to select the proper word from the cache entry and the cycle ends. If there is no match, or the valid bit is clear, a cache miss occurs and the instruction is fetched from external memory. This new instruction is automatically written into the cache entry, and the valid bit is set, unless the freeze cache bit has been set in the cache control register. Since the processor always prefetches instructions externally with long word, aligned bus cycles, both words of the entry will be updated, regardless of which word caused the miss.

NOTE

Data accesses are not cached, regardless of their associated address space.

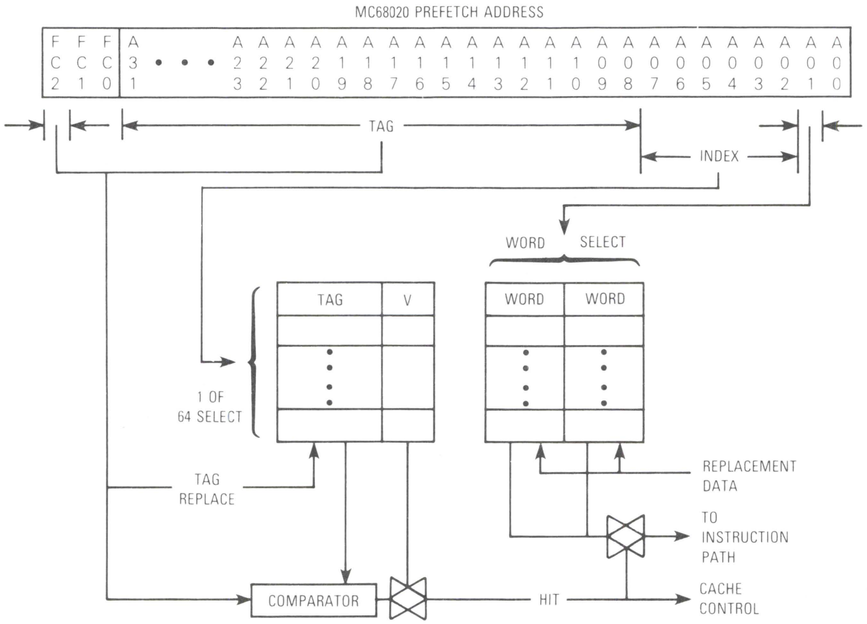

MC68020 PREFETCH ADDRESS
FC2 FC1 FC0
A31 • • • A23 A22 A21 A20 A19 A18 A17 A16 A15 A14 A13 A12 A11 A10 A09 A08 A07 A06 A05 A04 A03 A02 A01 A00
TAG
INDEX
WORD SELECT
TAG
V
WORD
WORD
1 OF 64 SELECT
TAG REPLACE
REPLACEMENT DATA
TO INSTRUCTION PATH
COMPARATOR
HIT
CACHE CONTROL

Figure 6-1. MC68020 On-Chip Cache Organization

6.2 CACHE RESET

During processor reset, the cache is cleared by resetting all of the valid bits. The cache control register (CACR), enable (E), and freeze (F) bits are also cleared.

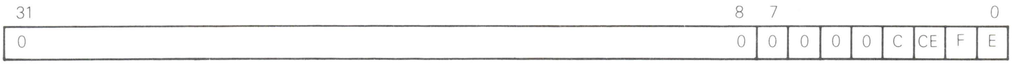
6.3 CACHE CONTROL

Only the MC68020 cache control circuitry can directly access the cache array, but the supervisor program can set bits in the CACR to exercise control over cache operations. The supervisor also has access to the cache address register (CAAR), which contains the address for a cache entry to be cleared.

System hardware can assert the cache disable ($\overline{\text{CDIS}}$) signal in order to disable the cache. The assertion of $\overline{\text{CDIS}}$ disables the cache, regardless of the state of the enable bits in the CACR. $\overline{\text{CDIS}}$ is primarily intended for use by in-circuit emulators.

6.3.1 Cache Control Register

The CACR, shown in Figure 6-2, is a 32-bit register than can be written or read by the MOVEC instruction or indirectly modified by a reset. Five of the bits (4–0) control the instruction cache. Bits 31–5 are reserved for Motorola definition. They are currently read as zeros and are ignored when written. For future compatibility, writes should not set these bits.



C = Clear Cache
CE = Clear Entry
F = Freeze Cache
E = Enable Cache

Figure 6-2. Cache Control Register

6.3.2 Clear Cache

CACR bit 3, the C bit, is set to clear all entries in the instruction cache. Operating systems and other software set this bit to clear instructions from the cache prior to a context switch. The processor clears all valid bits in the instruction cache at the time a MOVEC instruction loads a one into the C bit of the CACR. The C bit is always read as a zero.

6.3.3 Clear Entry In Cache

CACR bit 2, the CE bit, is set to clear an entry in the instruction cache. The index field of the CAAR, shown in Figure 6-3, coresponding to the index and long word select portion of an address, specifies the entry to be cleared. The processor clears only the specified long word by clearing the valid bit for the entry at the time a MOVEC instruction loads a one into the CE bit of the CACR, regardless of the states of the E and F bits. The CE bit is always read as a zero.

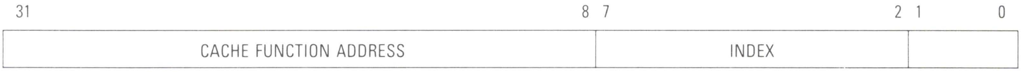


Figure 6-3. Cache Address Register

6.3.4 Freeze Cache

CACR bit 1, the F bit, is set to freeze the instruction cache. When the F bit is set, and a miss occurs in the instruction cache, the entry (or line) is not replaced. When the F bit is cleared to zero, a miss in the instruction cache causes the entry (or line) to be filled. A reset operation clears the F bit.

6.3.5 Enable Cache

CACR bit 0, the E bit, is set to enable the instruction cache. When it is cleared, the instruction cache is disabled. A reset operation clears the E bit. The supervisor normally enables the instruction cache, but it can clear E for system debugging or emulation, as required. Disabling the instruction cache does not flush the entries. If it is enabled again, the previously valid entries remain valid, and may be used.

6.3.6 Cache Address Register

The CAAR is a 32-bit register, shown in Figure 6-3. The index field (bits 7–2) contains the address for the "clear cache entry" operations. The bits of this field correspond to bit 7–2 of addresses; they specify the index and a long word of a cache line. Although only the index field is used currently, all 32 bits of the register are implemented and are reserved for use by Motorola.

SECTION 7
BUS OPERATION

This section provides a functional description of the bus, the signals that control it, and the bus cycles provided for data transfer operations. It also describes the error and halt conditions, bus arbitration, and the reset operation. Operation of the bus is the same whether the processor or an extenal device is the bus master; the names and descriptions of bus cycles are from the point-of-view of the bus master. For exact timing specifications, refer to **SECTION 12 ELECTRICAL CHARACTERISTICS**.

The MC68020 architecture supports byte, word, and long-word operands allowing access to 8-, 16-, and 32-bit data ports through the use of asynchronous cycles controlled by the data transfer and size acknowledge inputs ($\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$).

The MC68020 allows byte, word, and long-word operands to be located in memory on any byte boundary. For a misaligned transfer, more than one bus cycle may be required to complete the transfer regardless of port size. For a port less than 32 bits wide, multiple bus cycles may be required for an operand transfer due to either misalignment or a port width smaller than the operand size. Instruction words and their associated extension words must be aligned on word boundaries. The user should be aware that misalignment of word or long-word operands can cause the MC68020 to perform multiple bus cycles for the operand transfer, and, therefore, processor performance is optimized if word and long-word memory operands are aligned on word or long-word boundaries, respectively.

7.1 BUS TRANSFER SIGNALS

The bus transfers information between the MC68020 and an external memory, coprocessor, or peripheral device. External devices can accept or provide 8 bits, 16 bits, or 32 bits in parallel and must follow the handshake protocol described in this section. The maximum number of bits accepted or provided during a bus transfer is defined as the port width. The MC68020 contains an address bus that specifies the address for the transfer and a data bus that transfers the data. Control signals indicate the beginning of the cycle, the address space and the size of the transfer, and the type of cycle. The selected device then controls the length of the cycle with the signal(s) used to terminate the cycle. Strobe signals, one for the address bus and another for the data bus, indicate the validity of the address and provide timing information for the data.

The bus operates in an asynchronous mode for any port width. The bus and control input signals are internally synchronized to the MC68020 clock, introducing a delay. This delay is the time period required for the MC68020 to sample an input signal, synchronize the input to the internal clocks of the processor, and determine whether it is high or low. Figure 7-1 shows the relationship between the clock signal, a typical input, and its associated internal signal.

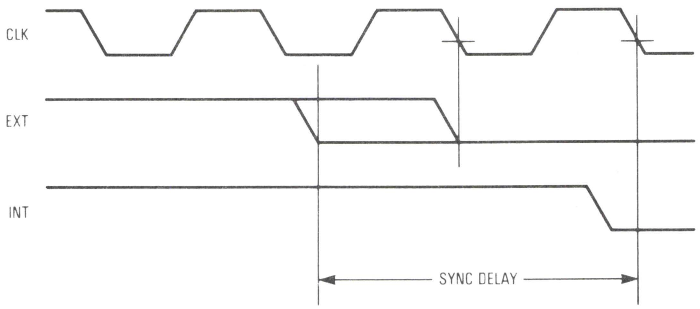


Figure 7-1. Relationship Between External and Internal Signals

Furthermore, for all inputs, the processor latches the level of the input during a sample window around the falling edge of the clock signal. This window is illustrated in Figure 7-2. To ensure that an input signal is recognized on a specific falling edge of the clock, that input must be stable during the sample window. If an input makes a transition during the window time period, the level recognized by the processor is not predictable; however, the processor always resolves the latched level to either a logic high or low before using it. In addition to meeting input setup and hold times for deterministic operation, all input signals must obey the protocols described in this section.

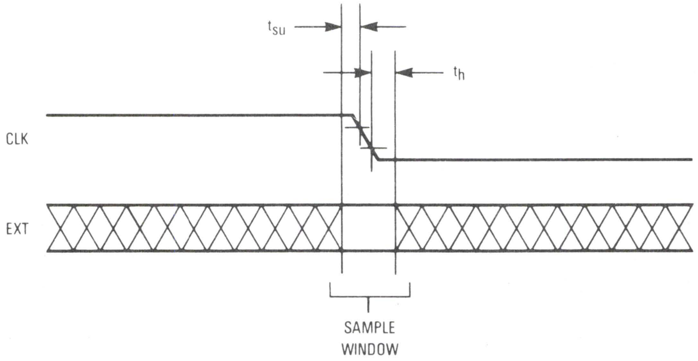


Figure 7-2. Input Sample Window

7.1.1 Bus Control Signals

The external cycle start ($\overline{\text{ECS}}$) signal is the earliest indication that the processor is initiating a bus cycle. The MC68020 initiates a bus cycle by driving the address, size, function code, read/write, and cache inhibit-out outputs and asserting $\overline{\text{ECS}}$. However, if the processor finds the required instruction in the on-chip cache, the processor aborts the cycle before asserting the address strobe ($\overline{\text{AS}}$). $\overline{\text{ECS}}$ can be used to initiate various timing sequences that are eventually qualified with $\overline{\mathrm{AS}}$. Qualification with $\overline{\mathrm{AS}}$ may be required since, in the case of an internal cache hit, a bus cycle may be aborted after $\overline{\mathrm{ECS}}$ has been asserted. The assertion of $\overline{\mathrm{AS}}$ ensures that the cycle has not been aborted by these internal conditions.

During the first extenal bus cycle of an operand transfer the operand cycle start ($\overline{\mathrm{OCS}}$) signal is asserted with $\overline{\mathrm{ECS}}$. When several bus cycles are required to transfer the entire operand, $\overline{\mathrm{OCS}}$ is asserted only at the beginning of the first external bus cycle. With respect to $\overline{\mathrm{OCS}}$, an ''operand'' is any entity required by the execution unit, whether a program or data item.

The function code signals (FC2–FC0) are also driven at the beginning of a bus cycle. These three signals select one of eight address spaces (refer to Table 4-1) to which the address applies. Five address spaces are presently defined. Of the remaining three, one is reserved for user definition and two are reserved by Motorola for future use. The function code signals are valid while $\overline{\mathrm{AS}}$ is asserted.

At the beginning of a bus cycle, the size signals (SIZ1, SIZ0) are driven along with $\overline{\mathrm{ECS}}$ and the function code signals. SIZ1 and SIZ0 indicate the number of bytes remaining to be transferred during an operand cycle (consisting of one or more bus cycles) or during a cache fill operation from a device with a port size that is less than 32 bits. Table 7-2 shows the encoding of SIZ1 and SIZ0. These signals are valid while $\overline{\mathrm{AS}}$ is asserted.

The read/write (R/$\overline{\mathrm{W}}$) signal determines the direction of the transfer during a bus cycle. This signal changes state, when required, at the beginning of a bus cycle, and is valid while $\overline{\mathrm{AS}}$ is asserted. R/$\overline{\mathrm{W}}$ only transitions when a write cycle is preceded by a read cycle or vice versa. The signal may remain low for two consecutive write cycles.

The read-modify-write cycle signal ($\overline{\mathrm{RMC}}$) is asserted at the beginning of the first bus cycle of a read-modify-write operation, and remains asserted until completion of the final bus cycle of the operation. The $\overline{\mathrm{RMC}}$ signal is guaranteed to be negated before the end of state zero for a bus cycle following a read-modify-write operation.

7.1.2 Address Bus

The address bus signals (A31–A0) define the address of the byte (or the most significant byte) to be transferred during a bus cycle. The processor places the address on the bus at the beginning of a bus cycle. The address is valid while $\overline{\mathrm{AS}}$ is asserted.

7.1.3 Address Strobe

The address strobe ($\overline{\mathrm{AS}}$) is a timing signal that indicates the validity of an address on the address bus and of many control signals. It is asserted one-half clock after the beginning of a bus cycle.

7.1.4 Data Bus

The data bus signals (D31–D0) comprise a bidirectional, nonmultiplexed parallel bus that contains the data being transferred to or from the processor. A read or write operation may transfer 8, 16, 24, or 32 bits of data (one, two, three, or four bytes) in one bus cycle. During a read cycle, the data is latched by the processor on the last falling edge of the clock for that bus cycle. For a write cycle, all 32 bits of the data bus are driven, regardless of the port width or operand size. The processor places the data on the data bus one-half clock cycle after AS is asserted in a write cycle.

7.1.5 Data Strobe

The data strobe (DS) is a timing signal that applies to the data bus. For a read cycle, the processor asserts DS to signal the external device to place data on the bus. It is asserted at the same time as AS during a read cycle. For a write cycle, DS signals to the external device that the data to be written is valid on the bus. The processor asserts DS one full clock cycle after the assertion of AS during a write cycle.

7.1.6 Data Buffer Enable

The data buffer enable signal (DBEN) can be used to enable external data buffers while data is present on the data bus. During a read operation, DBEN is asserted one clock cycle after the beginning of the bus cycle and is negated as DS is negated. In a write operation, DBEN is asserted at the time AS is asserted and is held active for the duration of the cycle.

7.1.7 Bus Cycle Termination Signals

During bus cycles, external devices assert the data transfer and size acknowledge signals (DSACK1 and/or DSACK0) as part of the bus protocol. During a read cycle, this signals the processor to terminate the bus cycle and to latch the data. During a write cycle, this indicates that the external device has successfully stored the data and that the cycle may terminate. These signals also indicate to the processor the size of the port for the bus cycle just completed, as shown in Table 7-1. Refer to **7.3.1 Read Cycle** for timing relationships of DSACK1 and DSACK0.

The bus error (BERR) signal is also a bus cycle termination indicator and can be used in the absence of DSACKx to indicate a bus error condition. It can also be asserted in conjunction with DSACKx to indicate a bus error condition, provided it meets the appropriate timing described in this section and in **SECTION 12 ELECTRICAL CHARACTERISTICS**. Additionally, the BERR and HALT signals can be asserted together to indicate a retry termination. Again, the BERR and HALT signals can be asserted simultaneously, in lieu of, or in conjunction with the DSACKx signals.

Table 7-1. DSACK Codes and Results

DSACK1	DSACK0	Result
H	H	Insert Wait States in Current Bus Cycle
H	L	Complete Cycle — Data Bus Port Size is 8 Bits
L	H	Complete Cycle — Data Bus Port Size is 16 Bits
L	L	Complete Cycle — Data Bus Port Size is 32 Bits

Finally, the autovector ($\overline{\text{AVEC}}$) signal can be used to terminate interrupt acknowledge cycles, indicating that the MC68020 should internally generate a vector number to locate an interrupt handler routine. $\overline{\text{AVEC}}$ is ignored during all other bus cycles.

7.2 DATA TRANSFER MECHANISM

The MC68020 architecture supports byte, word, and long-word operands allowing access to 8-, 16-, and 32-bit data ports through the use of asynchronous cycles controlled by the data transfer and size acknowledge inputs ($\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$). Byte, word, and long-word operands can be located on any byte boundary, but misaligned transfers may require additional bus cycles, regardless of port size.

7.2.1 Dynamic Bus Sizing

The MC68020 dynamically interprets the port size of the addressed device during each bus cycle, allowing operand transfers to or from 8-, 16-, and 32-bit ports. During an operand transfer cycle, the slave device signals its port size (byte, word, or long word) and indicates completion of the bus cycle to the processor through the use of the $\overline{\text{DSACKx}}$ inputs. Refer to Table 7-1 for $\overline{\text{DSACKx}}$ encodings and assertion results.

For example, if the processor is executing an instruction that reads a long-word operand from a long-word aligned address, it attempts to read 32 bits during the first bus cycle. (Refer to **7.2.2 Misaligned Operands** for the case of a word or byte address.) If the port responds that it is 32 bits wide, the MC68020 latches all 32 bits of data and continues with the next operation. If the port responds that it is 16 bits wide, the MC68020 latches the 16 bits of valid data and runs another bus cycle to obtain the other 16 bits. The operation for an 8-bit port is similar, but requires four read cycles. The addressed device uses the $\overline{\text{DSACKx}}$ signals to indicate the port width. For instance, a 32-bit device *always* returns $\overline{\text{DSACKx}}$ for a 32-bit port (regardless of whether the bus cycle is a byte, word, or long-word operation).

Dynamic bus sizing requires that the portion of the data bus used for a transfer to or from a particular port size be fixed. A 32-bit port must reside on data bus bits 0 through 31, a 16-bit port must reside on data bus bits 16 through 32, and an 8-bit port must reside on data bus bits 24 through 31. This requirement minimizes the number of bus cycles needed to transfer data to 8- and 16-bit ports and ensures that the MC68020 correctly transfers valid data. The MC68020 always attempts to transfer the maximum amount of data on all bus cycles; for a long word operation, it always assumes that the port is 32 bit wide when beginning the bus cycle.

The bytes of operands are designated as shown in Figure 7-3. The most significant byte of a long-word operand is OP0 and OP3 is the least significant byte. The two bytes of a word-length operand are OP2 (most significant) and OP3. The single byte of a byte-length operand is OP3. These designations are used in the figures and descriptions that follow.

Figure 7-4 shows the required organization of data ports on the MC68020 bus for 8-, 16-, and 32-bit devices. The four bytes shown in Figure 7-4 are connected through the internal data bus and data multiplexer to the external data bus. This path is the means through

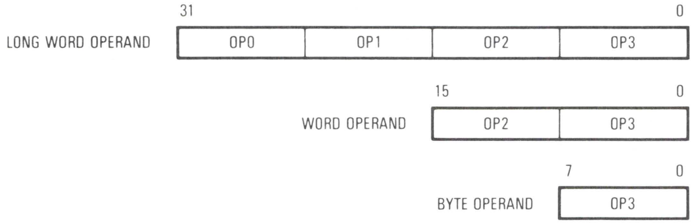


Figure 7-3. Internal Operand Representation

which the MC68020 supports dynamic bus sizing and operand misalignment. Refer to **7.2.2 Misaligned Operands** for the definition of misaligned operand. The data multiplexer establishes the necessary connections for different combinations of address and data sizes.

The multiplexer takes the four bytes of the 32-bit bus and routes them to their required positions. For example, OP0 can be routed to D31–D24, as would be the normal case, or it can be routed to any other byte position in order to support a misaligned transfer. The same is true for any of the operand bytes. The positioning of bytes is determined by the size (SIZ1 and SIZ0) and address A1 and A0) outputs.

The SIZ1 and SIZ0 outputs indicate the remaining number of bytes to be transferred during the current bus cycle, as shown in Table 7-2.

The number of bytes transferred during a write or read bus cycle is equal to or less than the size indicated by the SIZ1 and SIZ0 outputs, depending on port width and operand alignment. For example, during the first bus cycle of a long-word transfer to a word port, the size outputs indicate that four bytes are to be transferred although only two bytes are moved on that bus cycle.

The address lines A1 and A0 also affect operation of the data multiplexer. During an operand transfer, A31–A2 indicate the long-word base address of that portion of the operand to be accessed; A1 and A0 indicate the byte offset from the base. Table 7-3 shows the encodings of A1 and A0, and the corresponding byte offsets from the long-word base.

Table 7-4 lists the bytes required on the data bus for read cycles. The entries shown as OPn are portions of the requested operand that are read or written during that bus cycle and are defined by SIZ1, SIZ0, A1 and A0 for the bus cycle. The x bytes are "don't cares".

Table 7-2. Size Signal Encoding

SIZ1	SIZ0	Size
0	1	Byte
1	0	Word
1	1	3 Bytes
0	0	Long Word

Table 7-3. Address Offset Encodings

A1	A0	Offset
0	0	+0 Bytes
0	1	+1 Byte
1	0	+2 Bytes
1	1	+3 Bytes

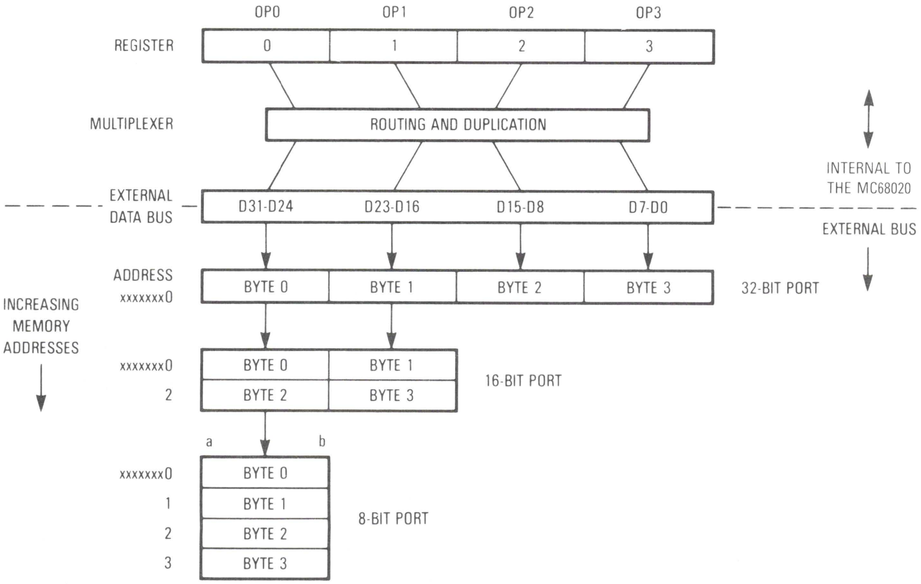

OP0
OP1
OP2
OP3
REGISTER
0
1
2
3
MULTIPLEXER
ROUTING AND DUPLICATION
INTERNAL TO THE MC68020
EXTERNAL DATA BUS
D31-D24
D23-D16
D15-D8
D7-D0
EXTERNAL BUS
ADDRESS xxxxxxx0
BYTE 0
BYTE 1
BYTE 2
BYTE 3
32-BIT PORT
INCREASING MEMORY ADDRESSES
xxxxxxx0
2
BYTE 0
BYTE 1
BYTE 2
BYTE 3
16-BIT PORT
a
b
xxxxxxx0
1
2
3
BYTE 0
BYTE 1
BYTE 2
BYTE 3
8-BIT PORT

Figure 7-4. MC68020 Interface to Various Port Sizes

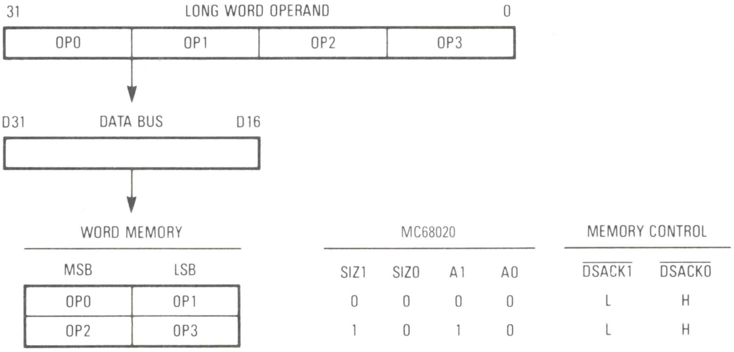
Table 7-5 lists the combinations of SIZ1, SIZ0, A1, and A0 and the corresponding pattern of the data transfer, for write cycles, from the internal multiplexer of the MC68020 to the external data bus.

Figure 7-5 shows the transfer of a long-word operand to a word port. In the first bus cycle, the MC68020 places the four operand bytes on the external bus. Since the address is long word aligned in this example, the multiplexer follows the pattern in the entry of Table 7-5 corresponding to SIZ1–SIZ0–A1–A0 = 0000. The port latches the data on bits D31–D16 of the data bus, asserts $\overline{\text{DSACK1}}$ ($\overline{\text{DSACK0}}$ remains negated), and the processor terminates the bus cycle. It then starts a new bus cycle with SIZ1–SIZ0–A1–A0 = 1010 to transfer the remaining 16 bits. The size signals indicate that a word remains to be transferred; A1 and A0 indicate that the word corresponds to an offset of two from the base address. The multiplexer follows the pattern corresponding to this configuration of the size and address signals and places the two least significant bytes of the long word on the word portion of the bus (D31–D16). The bus cycle transfers the remaining bytes to the word-size port. Figure 7-6 shows the timing of the bus transfer signals for this operation.

Figure 7-7 shows a word transfer to an 8-bit bus port. Like the preceding example, this example requires two bus cycles. Each bus cycle transfers a single byte. The size signals for the first cycle specify two bytes and, for the second cycle, one byte. Figure 7-8 shows the associated bus transfer signal timing.

Table 7-4. Data Bus Requirements for Read Cycles

Transfer Size	Size		Address		Long Word Port External Data Bytes Required				Word Port External Data Bytes Required		Byte Port External Data Bytes Required
	SIZ1	SIZ0	A1	A0	D31:D24	D23:D16	D15:D8	D7:D0	D31:D24	D23:D16	D31:D24
Byte	0	1	0	0	OP3				OP3		OP3
	0	1	0	1		OP3				OP3	OP3
	0	1	1	0			OP3		OP3		OP3
	0	1	1	1				OP3		OP3	OP3
Word	1	0	0	0	OP2	OP3			OP2	OP3	OP2
	1	0	0	1		OP2	OP3			OP2	OP2
	1	0	1	0			OP2	OP3	OP2	OP3	OP2
	1	0	1	1				OP2		OP2	OP2
3 Byte	1	1	0	0	OP1	OP2	OP3		OP1	OP2	OP1
	1	1	0	1		OP1	OP2	OP3		OP1	OP1
	1	1	1	0			OP1	OP2	OP1	OP2	OP1
	1	1	1	1				OP1		OP1	OP1
Long Word	0	0	0	0	OP0	OP1	OP2	OP3	OP0	OP1	OP0
	0	0	0	1		OP0	OP1	OP2		OP0	OP0
	0	0	1	0			OP0	OP1	OP0	OP1	OP0
	0	0	1	1				OP0		OP0	OP0


31
LONG WORD OPERAND
0
OP0
OP1
OP2
OP3
D31
DATA BUS
D16
WORD MEMORY
MC68020
MEMORY CONTROL

MSB	LSB
OP0	OP1
OP2	OP3

SIZ1	SIZ0	A1	A0	$\overline{DSACK1}$	$\overline{DSACK0}$
0	0	0	0	L	H
1	0	1	0	L	H

Figure 7-5. Example of Long-Word Transfer to Word Port

Table 7-5. MC68020 Internal to External Data Bus Multiplexer — Write Cycles

Transfer Size	Size		Address		External Data Bus Connection			
	SIZ1	SIZ0	A1	A0	D31:D24	D23:D16	D15:D8	D7:D0
Byte	0	1	x	x	OP3	OP3	OP3	OP3
Word	1	0	x	0	OP2	OP3	OP2	OP3
	1	0	x	1	OP2	OP2	OP3	OP2
3 Byte	1	1	0	0	OP1	OP2	OP3	OP0*
	1	1	0	1	OP1	OP1	OP2	OP3
	1	1	1	0	OP1	OP2	OP1	OP2
	1	1	1	1	OP1	OP1	OP2*	OP1
Long Word	0	0	0	0	OP0	OP1	OP2	OP3
	0	0	0	1	OP0	OP0	OP1	OP2
	0	0	1	0	OP0	OP1	OP0	OP1
	0	0	1	1	OP0	OP0	OP1*	OP0

*Due to the current implementation, this byte is output but never used.

x = don't care

NOTE: The OP tables on the external data bus refer to a particular byte of the operand that is written on that section of the data bus.

7.2.2 Misaligned Operands

Since operands may reside at any byte boundaries, they may be misaligned. A byte operand is properly aligned at any address; a word operand is misaligned at an odd address; a long word is misaligned at an address that is not evenly divisible by four. The MC68000, MC68008, and MC68010 implementations allow long-word transfers on odd-word boundaries but force exceptions if word or long-word operand transfers are attempted at odd-byte addresses. Although the MC68020 does not enforce any alignment restrictions for data operands (including PC-relative data addresses), some performance degradation occurs when additional bus cycles are required for long-word or word operands that are misaligned. For maximum performance, data items should be aligned on their natural boundaries. All instruction words and extension words must reside on word boundaries. Attempting to prefetch an instruction word at an odd address causes an address error exception.

Figure 7-9 shows the transfer of a long-word operand to an odd address in word-organized memory, which requires three bus cycles. For the first cycle, the size signals specify a long-word transfer, and the address offset (A2:A0) is 001. Since the port width is 16 bits, only the first byte of the long word is transferred. The slave device latches the byte and acknowledges the data transfer, indicating that the port is 16 bits wide. When the processor

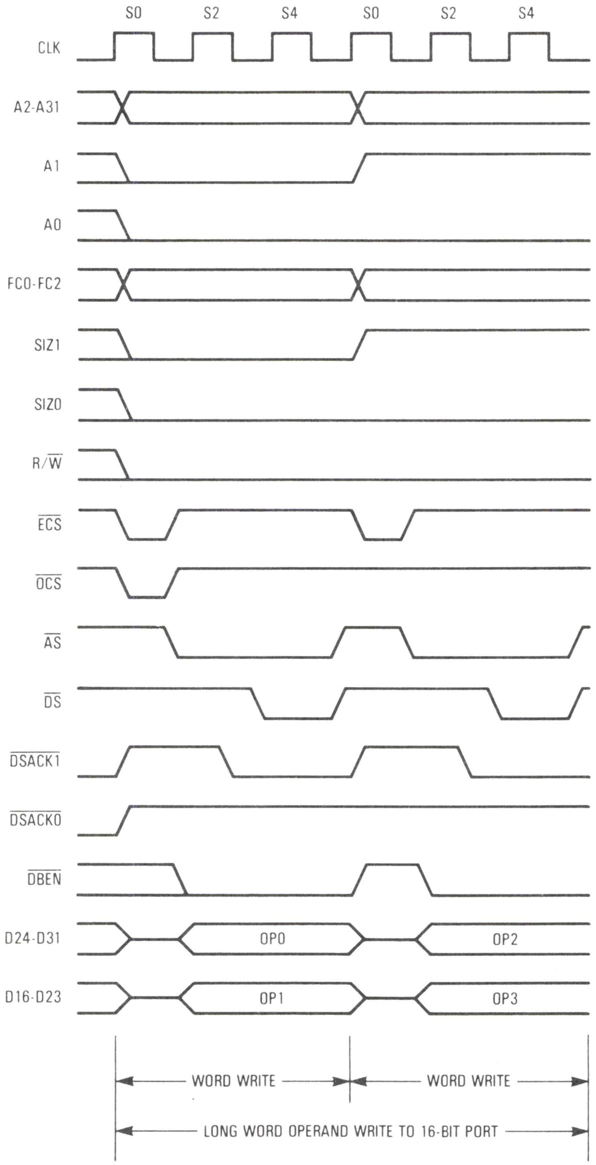
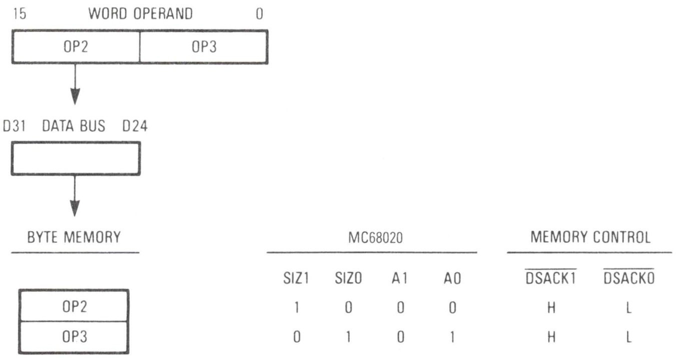

S0
S2
S4
S0
S2
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
SIZ0
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D24-D31
OP0
OP2
D16-D23
OP1
OP3
WORD WRITE
WORD WRITE
LONG WORD OPERAND WRITE TO 16-BIT PORT

Figure 7-6. Long-Word Operand Write Timing (16-Bit Data Port)

starts the second cycle, the size signals specify that three bytes remain to be transferred with an address offset (A2:A0) of 010. The next two bytes are transferred during this cycle. The processor then initiates the third cycle, with the size signals indicating one byte remaining to be transferred. The address offset (A2:A0) is now 100; the port latches the final



MC68020				MEMORY CONTROL	
SIZ1	SIZ0	A1	A0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
1	0	0	0	H	L
0	1	0	1	H	L

Figure 7-7. Example of Word Transfer to Byte Port

byte; and the operation is complete. Figure 7-10 shows the associated bus transfer signal timing. Figure 7-11 shows the equivalent operation for a data read cycle.

Figures 7-12 and 7-13 show a word transfer to an odd address in word-organized memory. This example is similar to the one shown in Figures 7-9 and 7-10 except that the operand is word sized and the transfer requires only two bus cycles. Figure 7-14 shows the equivalent operation for a data read cycle.

Figure 7-15 and 7-16 show an example of a long-word transfer to an odd address in long-word-organized memory. In this example, a long-word access is attempted beginning at the least significant byte of a long-word-organized memory. Only one byte can be transferred in the first bus cycle. The second bus cycle then consists of a three-byte access to a long-word boundary. Since the memory is long word organized, no further bus cycles are necessary. Figure 7-17 shows the equivalent operation for a read cycle.

7.2.3 Effects of Dynamic Bus Sizing and Operand Misalignment

The combination of operand size, operand alignment, and port size determine the number of bus cycles required to perform a particular memory access. Table 7-6 shows the number of bus cycles required for different operand sizes to different port sizes with all possible alignment conditions for write and read cycles.

This table shows that bus cycle throughput is significantly affected by port size and alignment. The MC68020 system designer and programmer should be aware of and account for these effects, particularly in time-critical applications.

Table 7-6 shows that the processor always prefetches instructions by reading a long word from a long-word address (A2:A0 = 000), regardless of port size or alignment. When the required instruction begins at an odd-word boundary, the processor attempts to fetch the entire 32 bits and loads both words into the instruction cache if possible, although the second one is the required word. Even if the instruction access is not cached, the entire

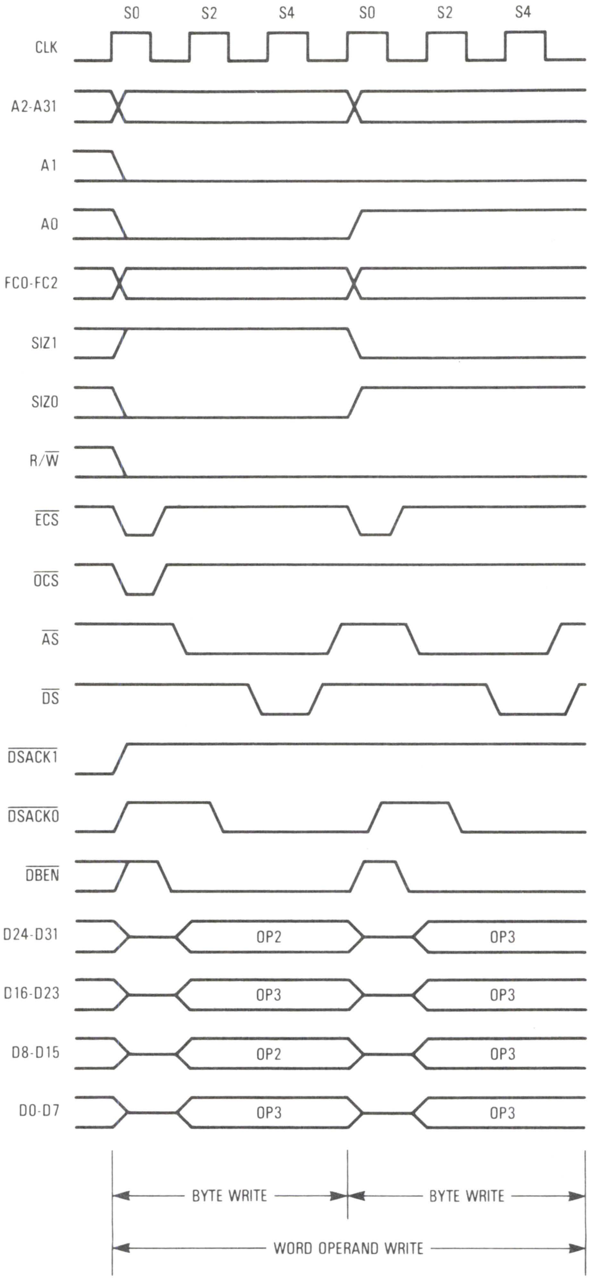
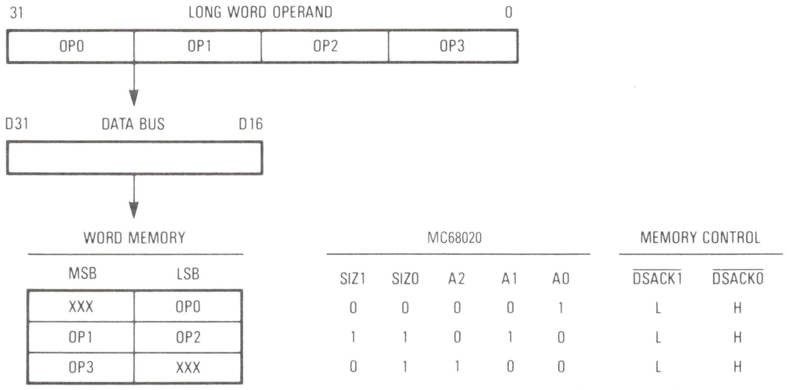

S0
S2
S4
S0
S2
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
SIZ0
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D24-D31
OP2
OP3
D16-D23
OP3
OP3
D8-D15
OP2
OP3
D0-D7
OP3
OP3
BYTE WRITE
BYTE WRITE
WORD OPERAND WRITE

Figure 7-8. Word Operand Write Timing (8-Bit Data Port)



MSB	LSB	SIZ1	SIZ0	A2	A1	A0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
XXX	OP0	0	0	0	0	1	L	H
OP1	OP2	1	1	0	1	0	L	H
OP3	XXX	0	1	1	0	0	L	H

Figure 7-9. Misaligned Long-Word Transfer to Word Port Example

Table 7-6. Memory Alignment and Port Size Influence on Write Bus Cycles

	Number of Bus Cycles			
A1/A0	**00**	**01**	**10**	**11**
Instruction*	1:2:4	N/A	N/A	N/A
Byte Operand	1:1:1	1:1:1	1:1:1	1:1:1
Word Operand	1:1:2	1:2:2	1:1:2	2:2:2
Long-Word Operand	1:2:4	2:3:4	2:2:4	2:3:4

Data Port Size 32-Bits: 16-Bits: 8-Bits
*Instruction prefetches are always two words from a long-word boundary.

32 bits are latched into an internal cache holding register from which the two instructions words can subsequently be referenced. Refer to **SECTION 10 INSTRUCTION EXECUTION TIMING** for a complete description of the cache holding register and pipeline operation.

7.2.4 Address, Size, and Data Bus Relationships

The data transfer examples show how the MC68020 drives data onto or receives data from the correct byte sections of the data bus. Table 7-7 shows the combinations of the size signals and address signals A1 and A0 that are used to generate byte enable signals for each of the four sections of the data bus for read and write cycles if the addressed device requires them. The port size also affects the generation of these enable signals as shown in the table. The four columns on the right correspond to the four byte enable signals. Letters B, W, and L, refer to port sizes: B for 8-bit ports, W for 16-bit ports, and L for 32-bit ports. The letters B, W, and L implies that the byte enable signal should be true for that port size. A dash (—) implies that the byte enable signal does not apply.

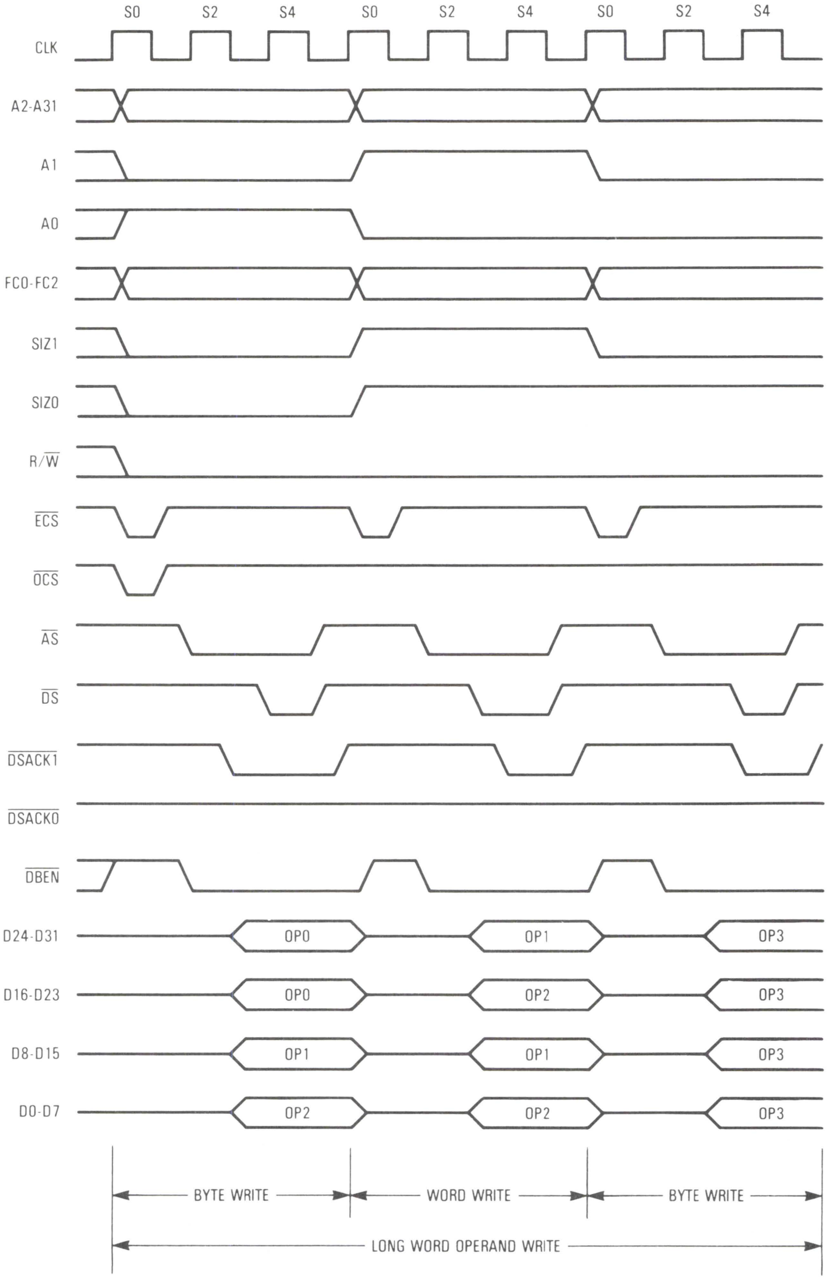
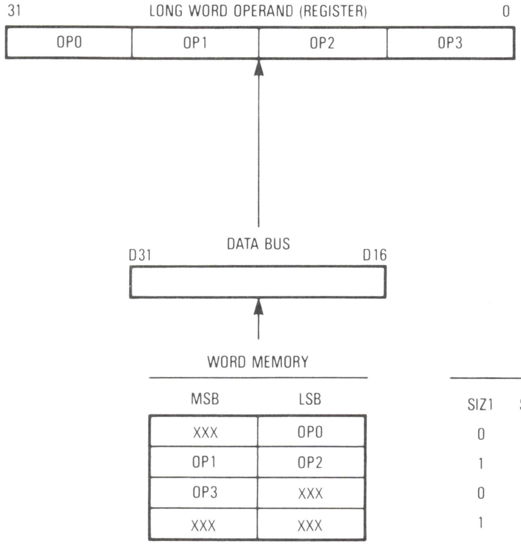
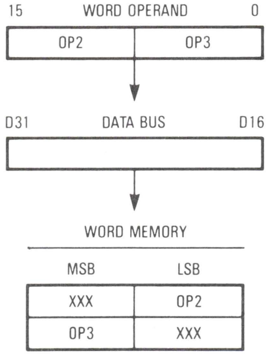

S0
S2
S4
S0
S2
S4
S0
S2
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
SIZ0
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D24-D31
D16-D23
D8-D15
D0-D7
OP0
OP1
OP3
OP0
OP2
OP3
OP1
OP1
OP3
OP2
OP2
OP3
BYTE WRITE
WORD WRITE
BYTE WRITE
LONG WORD OPERAND WRITE

Figure 7-10. Misaligned Long-Word Transfer to Word Port


31
LONG WORD OPERAND (REGISTER)
0
OP0
OP1
OP2
OP3
D31
DATA BUS
D16
WORD MEMORY
MSB
LSB
XXX
OP0
OP1
OP2
OP3
XXX
XXX
XXX

MC68020					MEMORY CONTROL	
SIZ1	SIZ0	A2	A1	A0	$\overline{DSACK1}$	$\overline{DSACK0}$
0	0	0	0	1	L	H
1	1	0	1	0	L	H
0	1	1	0	0	L	H
1	0	1	1	0	L	H

Figure 7-11. Misaligned Long-Word Transfer from Word Port Example


15
WORD OPERAND
0
OP2
OP3
D31
DATA BUS
D16
WORD MEMORY
MSB
LSB
XXX
OP2
OP3
XXX

MC68020					MEMORY CONTROL	
SIZ1	SIZ0	A2	A1	A0	$\overline{DSACK1}$	$\overline{DSACK0}$
1	0	0	0	1	L	H
0	1	0	1	0	L	H

Figure 7-12. Misaligned Word Transfer to Word Port Example

The MC68020 always drives all sections of the data bus because at the start of a write cycle, the bus controller does not know the port size.

The table shows that the MC68020 transfers the number of bytes specified by the size signals to or from the specified address unless the operand is misaligned, or unless the number of bytes is greater than the port width. In these cases, the device transfers the

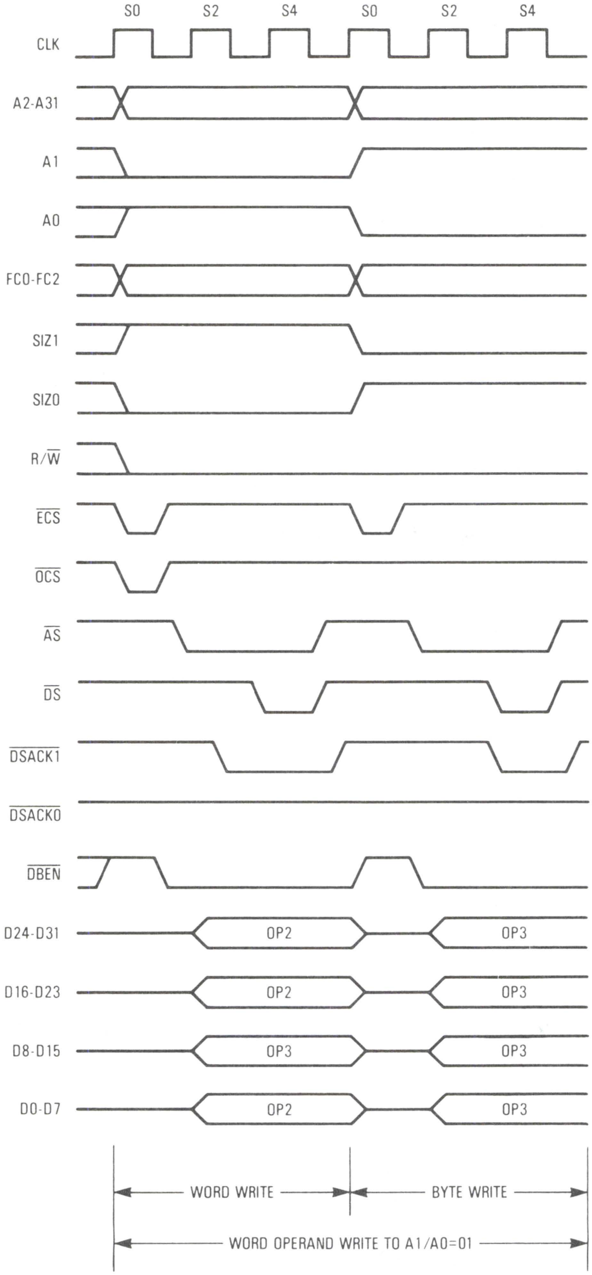
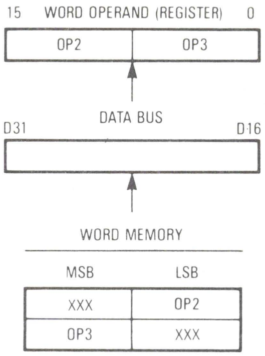
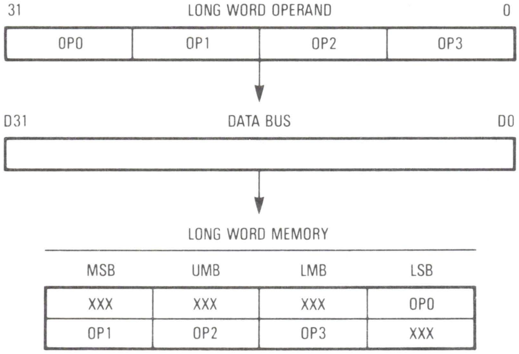

S0
S2
S4
S0
S2
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
SIZ0
R/W̄
ECS̄
OCS̄
AS̄
DS̄
DSACK1̄
DSACK0̄
DBEN̄
D24-D31
D16-D23
D8-D15
D0-D7
OP2
OP3
OP2
OP3
OP3
OP3
OP2
OP3
WORD WRITE
BYTE WRITE
WORD OPERAND WRITE TO A1/A0=01

Figure 7-13. Misaligned Word Transfer to Word Port


15
WORD OPERAND (REGISTER)
0
OP2
OP3
DATA BUS
D31
D16
WORD MEMORY
MSB
LSB
XXX
OP2
OP3
XXX

MC68020					MEMORY CONTROL	
SIZ1	SIZ0	A2	A1	A0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
1	0	0	0	1	L	H
0	1	0	1	0	L	H

Figure 7-14. Example of Misaligned Word Transfer from Word Bus


31
LONG WORD OPERAND
0
OP0
OP1
OP2
OP3
D31
DATA BUS
D0
LONG WORD MEMORY
MSB
UMB
LMB
LSB
XXX
XXX
XXX
OP0
OP1
OP2
OP3
XXX

MC68020					MEMORY CONTROL	
SIZ1	SIZ0	A2	A1	A0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
0	0	0	1	1	L	L
1	1	1	0	0	L	L

Figure 7-15 Misaligned Long-Word Transfer to Long-Word Port

greatest number of bytes possible for the port. For example, if the size is four bytes and the address offset (A1:A0) is 01, a 32-bit slave can only receive three bytes in the current bus cycle. A 16- or 8-bit slave can only receive one byte. The table defines the byte enables for all port sizes. Byte data strobes can be otained by combining the enable signals with the data strobe signal. Devices residing on 8-bit ports can use the data strobe by itself since there is only one valid byte for every transfer. These enable or strobe signals select only the bytes required for write or read cycles. The other bytes are not selected, which prevents incorrect accesses in sensitive areas such as I/O.

Figure 7-18 shows a logic diagram for one method for generating byte data enable signals for 16- and 32-bit ports from the size and address encodings and the read/write signal.

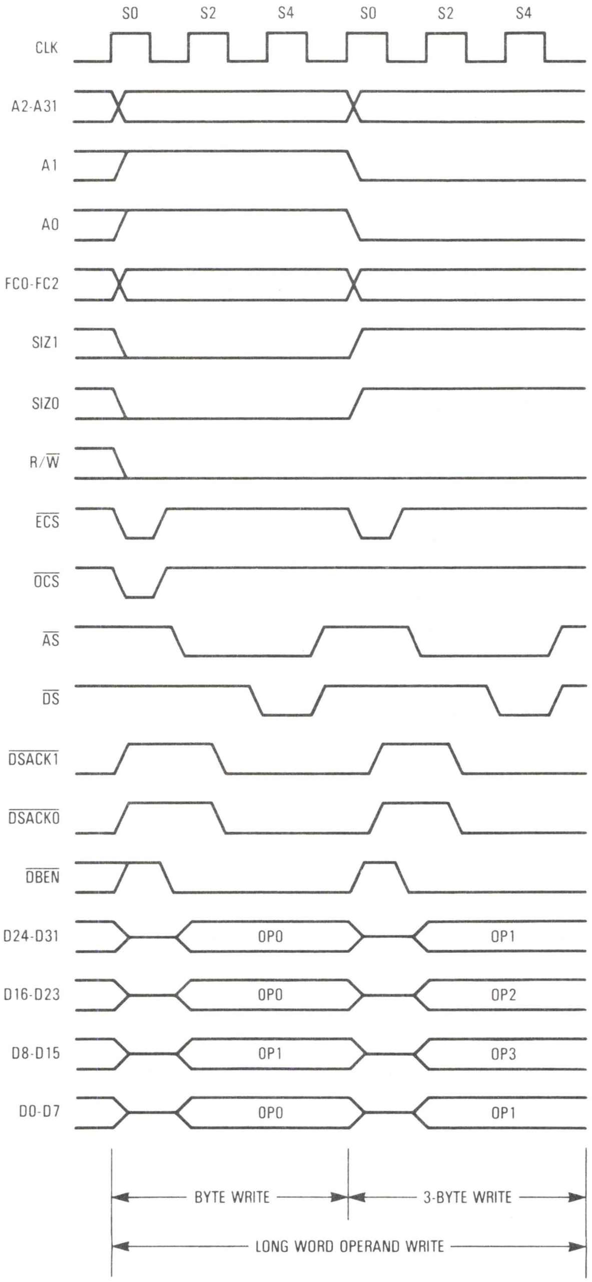
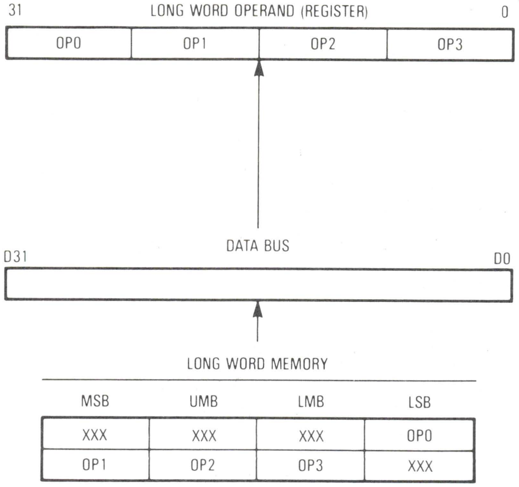

S0
S2
S4
S0
S2
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
SIZ0
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D24-D31
D16-D23
D8-D15
D0-D7
OP0
OP1
OP0
OP2
OP1
OP3
OP0
OP1
BYTE WRITE
3-BYTE WRITE
LONG WORD OPERAND WRITE

Figure 7-16 Misaligned Write Cycles to Long-Word Port


31
LONG WORD OPERAND (REGISTER)
0
OP0
OP1
OP2
OP3
DATA BUS
D31
D0
LONG WORD MEMORY
MSB
UMB
LMB
LSB
XXX
XXX
XXX
OP0
OP1
OP2
OP3
XXX

MC68020					MEMORY CONTROL	
SIZ1	SIZ0	A2	A1	A0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
0	0	0	1	1	L	L
1	1	1	0	0	L	L

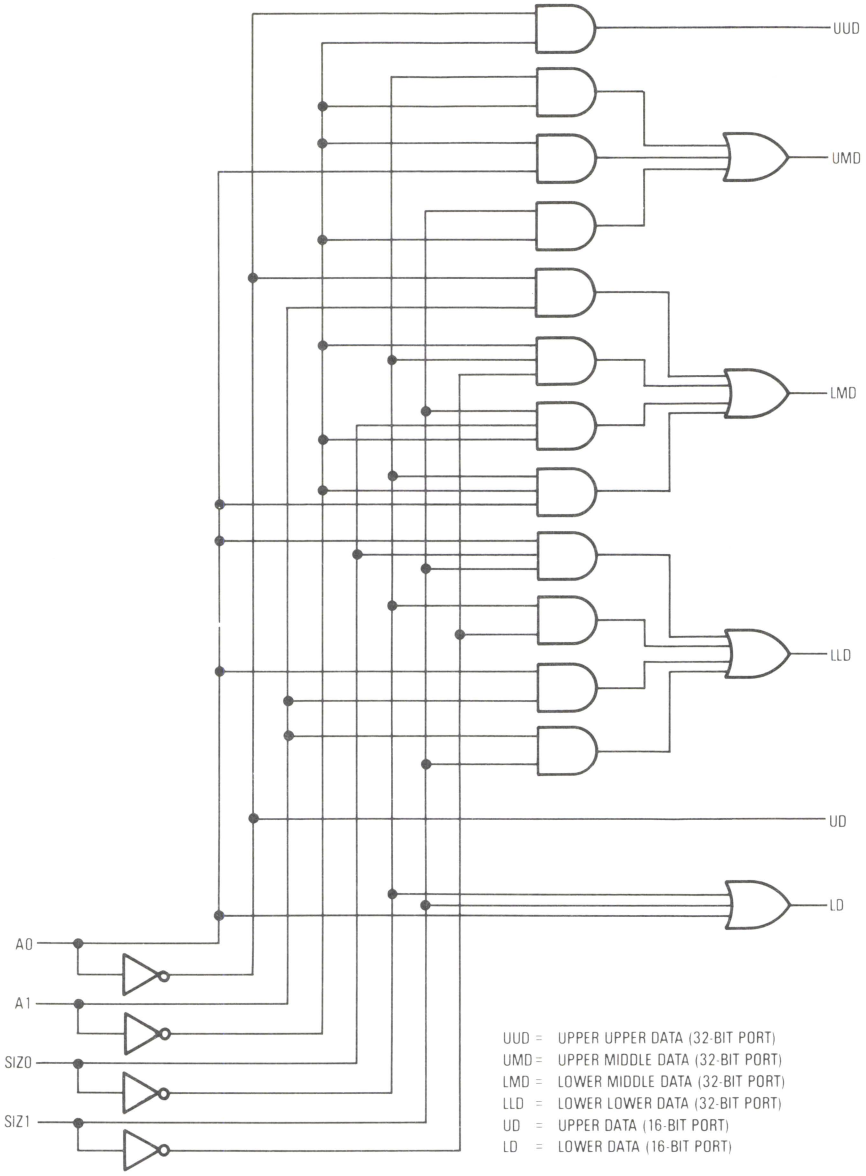
Figure 7-17. Misaligned Long-Word Transfer from Long-Word Bus

Table 7-7. Data Bus Write Enable Signals for Byte, Word, and Long-Word Ports

Transfer Size	SIZ1	SIZ0	A1	A0	Data Bus Active Sections Byte (B) – Word (W) – Long Word (L) Ports			
					D31:D24	D23:D16	D15:D8	D7:D0
Byte	0	1	0	0	B W L	—	—	—
	0	1	0	1	B	W L	—	—
	0	1	1	0	B W	—	L	—
	0	1	1	1	B	W	—	L
Word	1	0	0	0	B W L	W L	—	—
	1	0	0	1	B	W L	L	—
	1	0	1	0	B W	W	L	L
	1	0	1	1	B	W	—	L
3 Byte	1	1	0	0	B W L	W L	L	—
	1	1	0	1	B	W L	L	L
	1	1	1	0	B W	W	L	L
	1	1	1	1	B	W	—	L
Long Word	0	0	0	0	B W L	W L	L	L
	0	0	0	1	B	W L	L	L
	0	0	1	0	B W	W	L	L
	0	0	1	1	B	W	—	L

7.2.5 Cache Interactions

The organization and requirements of the on-chip instruction cache affects the interpretation of the $\overline{\text{DSACKx}}$ signals. Since the MC68020 attempts to load all instructions into the on-chip cache, the bus may operate differently when caching is enabled. Specifically, on read cycles that terminate normally, the low-order address signals (A1 and A0) and the size signals do not apply.


UUD
UMD
LMD
LLD
UD
LD
A0
A1
SIZ0
SIZ1
UUD = UPPER UPPER DATA (32-BIT PORT)
UMD = UPPER MIDDLE DATA (32-BIT PORT)
LMD = LOWER MIDDLE DATA (32-BIT PORT)
LLD = LOWER LOWER DATA (32-BIT PORT)
UD = UPPER DATA (16-BIT PORT)
LD = LOWER DATA (16-BIT PORT)

NOTE: These select lines can be combined with the address decode circuitry, or all can be generated within the same programmed array logic unit.

Figure 7-18. Byte Data Select Generation for 16- and 32-Bit Ports

The cache can also affect the assertion of the address strobe signal ($\overline{\text{AS}}$) and the operation of a read cycle. The search of the cache by the processor begins when the sequencer requires an instruction. At this time, the bus controller may also initiate an external bus cycle in case the requested item is not resident in the instruction cache. If the bus is not occupied with another read or write cycle, the bus controller asserts the ECS signal (and the OCS signal, if appropriate). If an internal cache hit occurs, the external cycle aborts, the AS is not asserted. This makes it possible to have ECS asserted on multiple consecutive clock cycles. Notice that there is a minimum time specified from the negation of ECS to the next assertion of ECS (refer to **SECTION 12 ELECTRICAL CHARACTERISTICS**).

Instruction prefetches can occur every other clock so that if, after an aborted cycle due to an instruction cache hit, the bus controller asserts ECS on the next clock, this second cycle is for a data fetch. Note that if the bus controller is executing other cycles, these aborted cycles due to cache hits may not be seen externally.

7.2.6 Bus Operation

The MC68020 bus is used in an asynchonous manner. The external devices connected to the bus can operate at clock frequencies different from the clock for the MC68020. Bus operation uses the handshake lines (AS, DS, DSACK1, DSACK0, BERR, and HALT) to control data transfers. AS signals the start of a bus cycle, and DS is used as a condition for valid data on a write cycle. Decoding the size outputs and lower address lines A1 and A0 provides strobes that select the active portion of the data bus. The slave device (memory or peripheral) then responds by placing the requested data on the correct portion of the data bus for a read cycle or latching the data on a write cycle, and asserting the DSACK1/DSACK0 combination that corresponds to the port size to terminate the cycle. If no slave responds or the access is invalid, external control logic asserts the BERR, or BERR and HALT signal(s) to abort or retry the bus cycle, respectively.

The DSACKx signals can be asserted before the data from a slave device is valid on a read cycle. The length of time that DSACKx may precede data is given by parameter #31, and it must be met in any asynchronous system to insure that valid data is latched into the processor. (Refer to **SECTION 12 ELECTRICAL CHARACTERISTICS** for timing parameters.) Notice that no maximum time is specified from the assertion of AS to the assertion of DSACKx. Although the processor can transfer data in a minimum of three clock cycles when the cycle is terminated with DSACKx, the processor inserts wait cycles in clock period increments until DSACKx is recognized.

The BERR and/or HALT signals can be asserted after the DSACKx signal(s) is asserted. BERR and/or HALT must be asserted within the time given as parameter #48, after DSACKx is asserted in any asynchronous system. If this maximum delay time is violated, the processor may exhibit erratic behavior.

7.2.7 Synchronous Operation with DSACKx

Although cycles terminated with the DSACKx signals are classified as "asynchronous", cycles terminated with DSACKx can also operate synchronously in that signals are interpreted relative to clock edges. The devices that use these cycles must synchronize the response to the MC68020 clock in order to be synchronous. Since they terminate bus cycles with the data transfer and size acknowledge signals (DSACKx), the dynamic bus sizing capabilities of the MC68020 are available. In addition, the minimum cycle time for these cycles is also three clocks.

To support those systems that use the system clock to generate DSACKx and other asynchronous inputs, the asynchronous input setup time (parameter #47A), and the asynchronous input hold time (parameter #47B) are given. If the setup and hold times are met for the assertion or negation of a signal, such as DSACKx, the processor can be guaranteed to recognize that signal level on that specific falling edge of the system clock. If the assertion of DSACKx is recognized on a particular falling edge of the clock, valid data is latched into the processor (for a read cycle) on the next falling clock edge provided that the data meets the data setup time (parameter #27). In this case, parameter #31 for asynchronous operation can be ignored. The timing parameters referred to are described in **SECTION 12 ELECTRICAL CHARACTERISTICS**. Note that if a system asserts DSACKx for the required window around the falling edge of S2 and obeys the proper bus protocol by maintaining DSACKx (and/or BERR/HALT) until and throughout the clock edge that negates AS (with the appropriate asynchronous input hold time specified by parameter #47B), no wait states are inserted. The bus cycle runs at its maximum speed for bus cycles terminated with DSACKx of three clocks per cycle.

To assure proper operation in a synchronous system when BERR or BERR and HALT is asserted after DSACKx, BERR (and HALT) must meet the appropriate setup time (parameter #27A) prior to the falling clock edge one clock cycle afterDSACKx is recognized. This setup time is critical, and the MC68020 may exhibit erratic behavior if it is violated.

When operating synchronously, the data-in setup and hold times for synchronous cycles may be used instead of the timing requirements for data relative to the DS signal.

7.3 DATA TRANSFER CYCLES

The transfer of data between the processor and other devices involves the following signals:

- Address Bus A31 through A0
- Data Bus D31 through D0
- Control Signals

The address and data buses are both parallel, nonmultiplexed buses. The bus master moves data on the bus by issuing control signals, and the bus uses a handshake protocol to ensure correct movement of the data. In all bus cycles, the bus master is responsible for de-skewing all signals it issues at both the start and the end of the cycle. In addition, the bus master is responsible for de-skewing the acknowledge and data signals from the slave devices. The following paragraphs define read, write, and read-modify-write cycle operations.

Each of the bus cycles is defined as a succession of states. These states apply to the bus operation and are different from the processor states described in **SECTION 4 PROCESSING STATES**. The clock cycles used in the descriptions and timing diagrams of data transfer cycles are independent of the clock frequency. Bus operations are described in terms of external bus states.

7.3.1 Read Cycle

During a read cycle, the processor receives data from a memory, coprocessor, or peripheral device. If the instruction specifies a long-word operation, the MC68020 attempts to read four bytes at once. For a word operation, it attempts to read two bytes at once, and for a byte operation, one byte. For some operations, the processor requests a three-byte transfer. The processor properly positions each byte internally. The section of the data bus from which each byte is read depends on the operand size, address signals A1 and A0, and the port size. Refer to **7.2.1 Dynamic Bus Sizing**, and **7.2.2 Misaligned Operands** for more information on dynamic bus sizing and misaligned operands.

Figure 7-19 is a flowchart of a long-word read cycle. Figure 7-20 is a flowchart of a byte read cycle. The following figures show functional read cycle timing diagrams specified in terms of clock periods. Figure 7-21 corresponds to byte and word read cycles from a 32-bit port. Figure 7-22 corresponds to a long-word read cycle from an 8-bit port. Figure 7-23 also applies to a long-word read, but from a 16-bit port.

State 0

The read cycle starts in state 0 (S0). The processor drives external cycle start ($\overline{\text{ECS}}$) low, indicating the beginning of an external cycle. When the cycle is the first external cycle of a read operation, operand cycle start ($\overline{\text{OCS}}$) is driven low at the same time. During S0, the processor places a valid address on the address bus (A31–A0) and valid function codes on FC2–FC0. The function codes select the address space for the cycle. The processor drives read/write (R/$\overline{\text{W}}$) high for a read cycle, and drives data buffer enable ($\overline{\text{DBEN}}$) inactive to disable the data buffers. Size signals SIZ1 and SIZ0 become valid, indicating the number of bytes requested to be transferred.

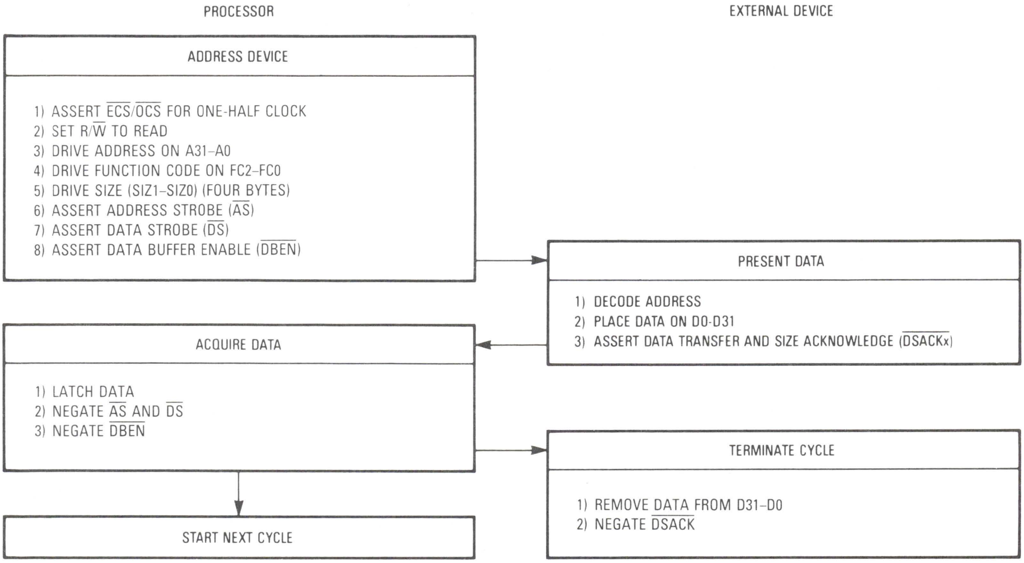

PROCESSOR
EXTERNAL DEVICE
ADDRESS DEVICE
1) ASSERT ECS/OCS FOR ONE-HALF CLOCK
2) SET R/W TO READ
3) DRIVE ADDRESS ON A31–A0
4) DRIVE FUNCTION CODE ON FC2–FC0
5) DRIVE SIZE (SIZ1–SIZ0) (FOUR BYTES)
6) ASSERT ADDRESS STROBE (AS)
7) ASSERT DATA STROBE (DS)
8) ASSERT DATA BUFFER ENABLE (DBEN)
PRESENT DATA
1) DECODE ADDRESS
2) PLACE DATA ON D0-D31
3) ASSERT DATA TRANSFER AND SIZE ACKNOWLEDGE (DSACKx)
ACQUIRE DATA
1) LATCH DATA
2) NEGATE AS AND DS
3) NEGATE DBEN
TERMINATE CYCLE
1) REMOVE DATA FROM D31–D0
2) NEGATE DSACK
START NEXT CYCLE

Figure 7-19. Long-Word Read Cycle Flowchart

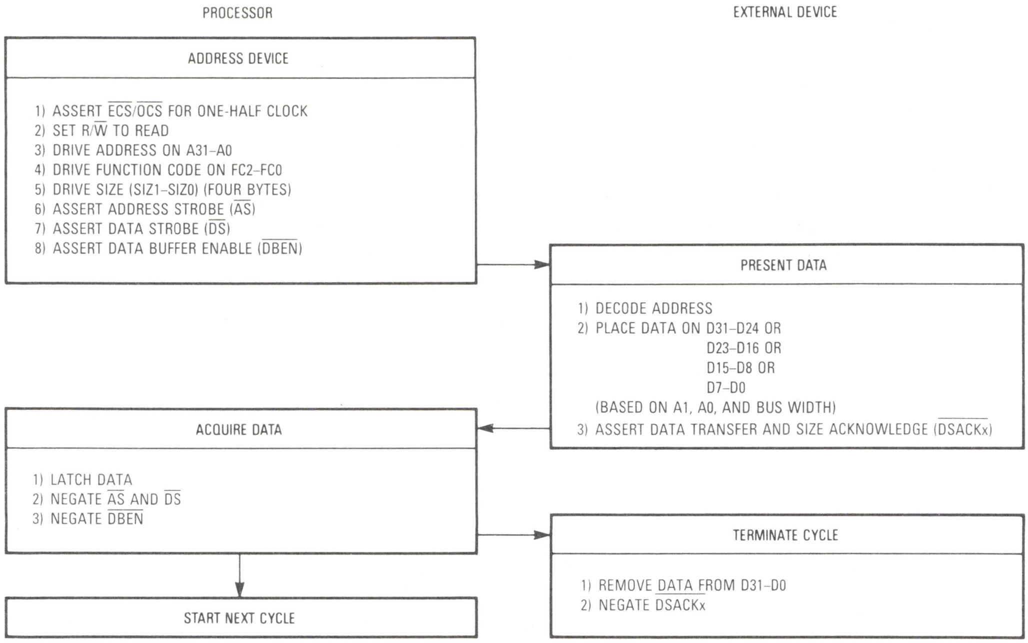

PROCESSOR
EXTERNAL DEVICE
ADDRESS DEVICE
1) ASSERT ECS/OCS FOR ONE-HALF CLOCK
2) SET R/W TO READ
3) DRIVE ADDRESS ON A31–A0
4) DRIVE FUNCTION CODE ON FC2–FC0
5) DRIVE SIZE (SIZ1–SIZ0) (FOUR BYTES)
6) ASSERT ADDRESS STROBE (AS)
7) ASSERT DATA STROBE (DS)
8) ASSERT DATA BUFFER ENABLE (DBEN)
PRESENT DATA
1) DECODE ADDRESS
2) PLACE DATA ON D31–D24 OR D23–D16 OR D15–D8 OR D7–D0
(BASED ON A1, A0, AND BUS WIDTH)
3) ASSERT DATA TRANSFER AND SIZE ACKNOWLEDGE (DSACKx)
ACQUIRE DATA
1) LATCH DATA
2) NEGATE AS AND DS
3) NEGATE DBEN
TERMINATE CYCLE
1) REMOVE DATA FROM D31–D0
2) NEGATE DSACKx
START NEXT CYCLE

Figure 7-20. Byte Read Cycle Flowchart

State 1

One-half clock later, in state 1 (S1), the processor asserts the address strobe ($\overline{\text{AS}}$) indicating that the address on the address bus is valid. The processor asserts the data strobe ($\overline{\text{DS}}$) also during S1. In addition, the $\overline{\text{ECS}}$ (and $\overline{\text{OCS}}$, if asserted) signal is negated during S1.

State 2

During state 2 (S2), the processor asserts $\overline{\text{DBEN}}$ to enable external data buffers. The selected device uses R/$\overline{\text{W}}$, SIZ1–SIZ0, A1–A0, and $\overline{\text{DS}}$ to place its information on the data bus. Any or all of the bytes (D31–D24, D23–D16, D15–D8, and D7–D0) are selected by the size signals and A1–A0. Concurrently, the selected device asserts the data transfer and size acknowledge ($\overline{\text{DSACKx}}$) signals.

State 3

As long as at least one of the $\overline{\text{DSACKx}}$ signals is recognized by the end of S2 (meeting the asynchronous input setup time requirement), data is latched on the next falling edge of the clock, and the cycle terminates. If $\overline{\text{DSACKx}}$ is not recognized by the start of state 3 (S3), the processor inserts wait states instead of proceeding to states 4 and 5. In order to ensure that wait states are inserted, both $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ must remain negated throughout the asynchronous input setup and hold times around the end of S2. If wait states are added, the processor continues to sample the $\overline{\text{DSACKx}}$ signals on the falling edges of the clock until one is recognized.

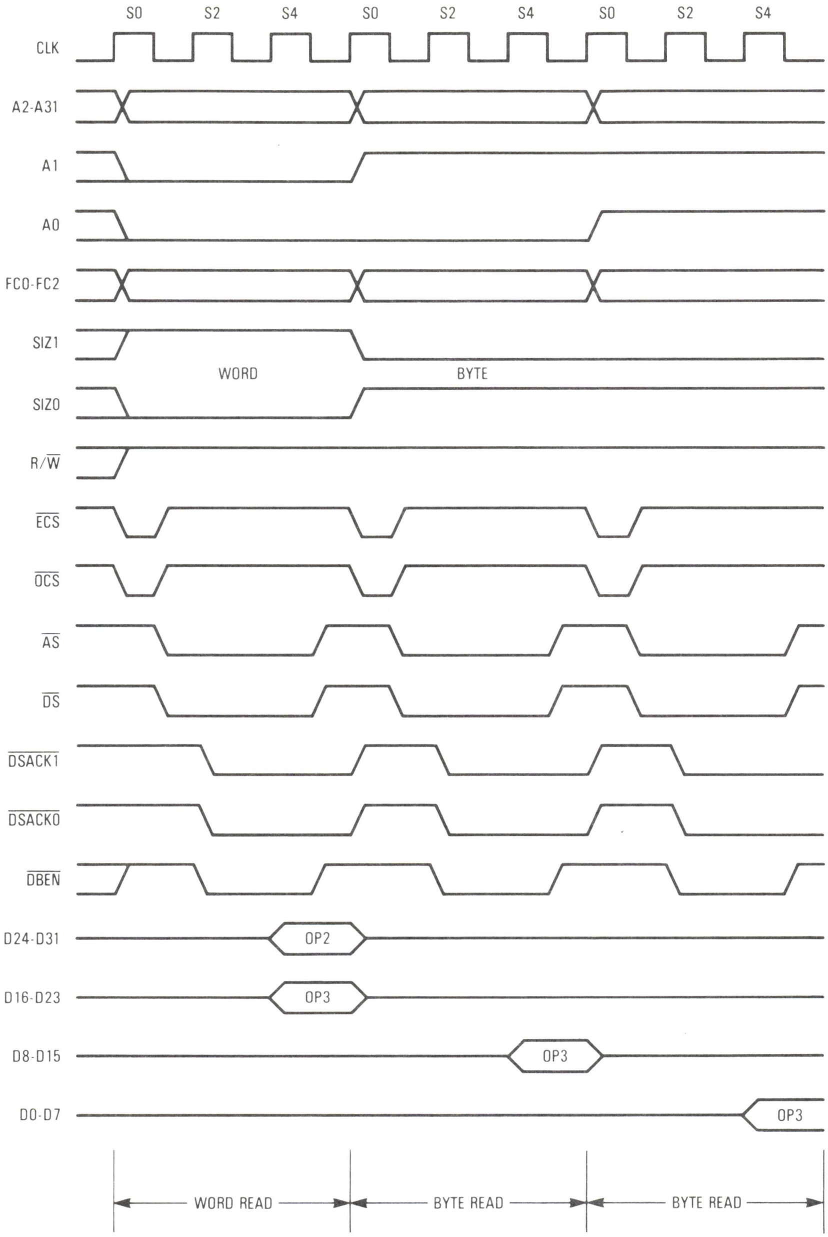

S0
S2
S4
S0
S2
S4
S0
S2
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
WORD
BYTE
SIZ0
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D24-D31
OP2
D16-D23
OP3
D8-D15
OP3
D0-D7
OP3
WORD READ
BYTE READ
BYTE READ

Figure 7-21. Byte and Word Read Cycles — 32-Bit Port

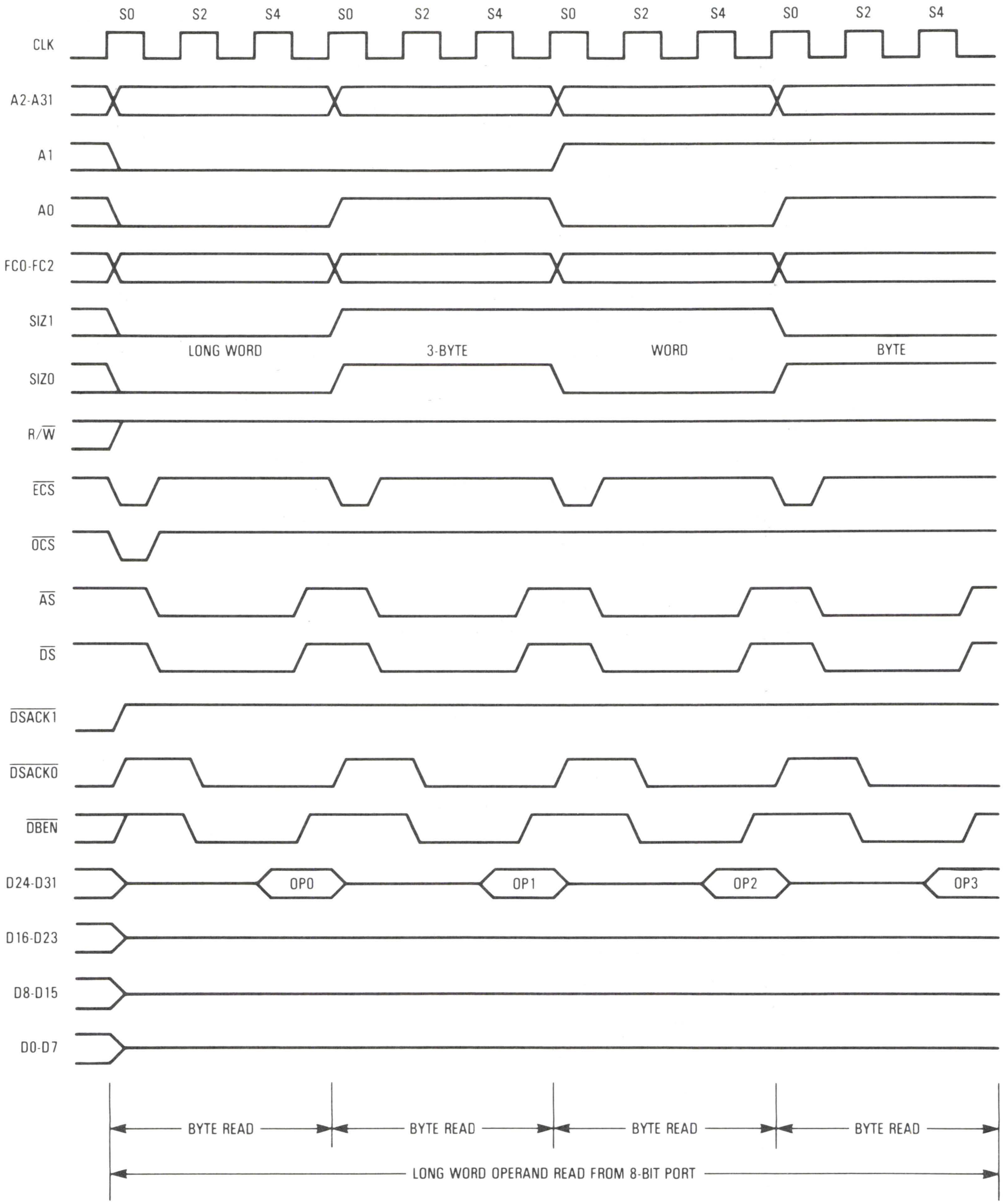

S0
S2
S4
S0
S2
S4
S0
S2
S4
S0
S2
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
LONG WORD
3-BYTE
WORD
BYTE
SIZ0
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D24-D31
OP0
OP1
OP2
OP3
D16-D23
D8-D15
D0-D7
BYTE READ
BYTE READ
BYTE READ
BYTE READ
LONG WORD OPERAND READ FROM 8-BIT PORT

Figure 7-22. Long-Word Read — 8-Bit Port

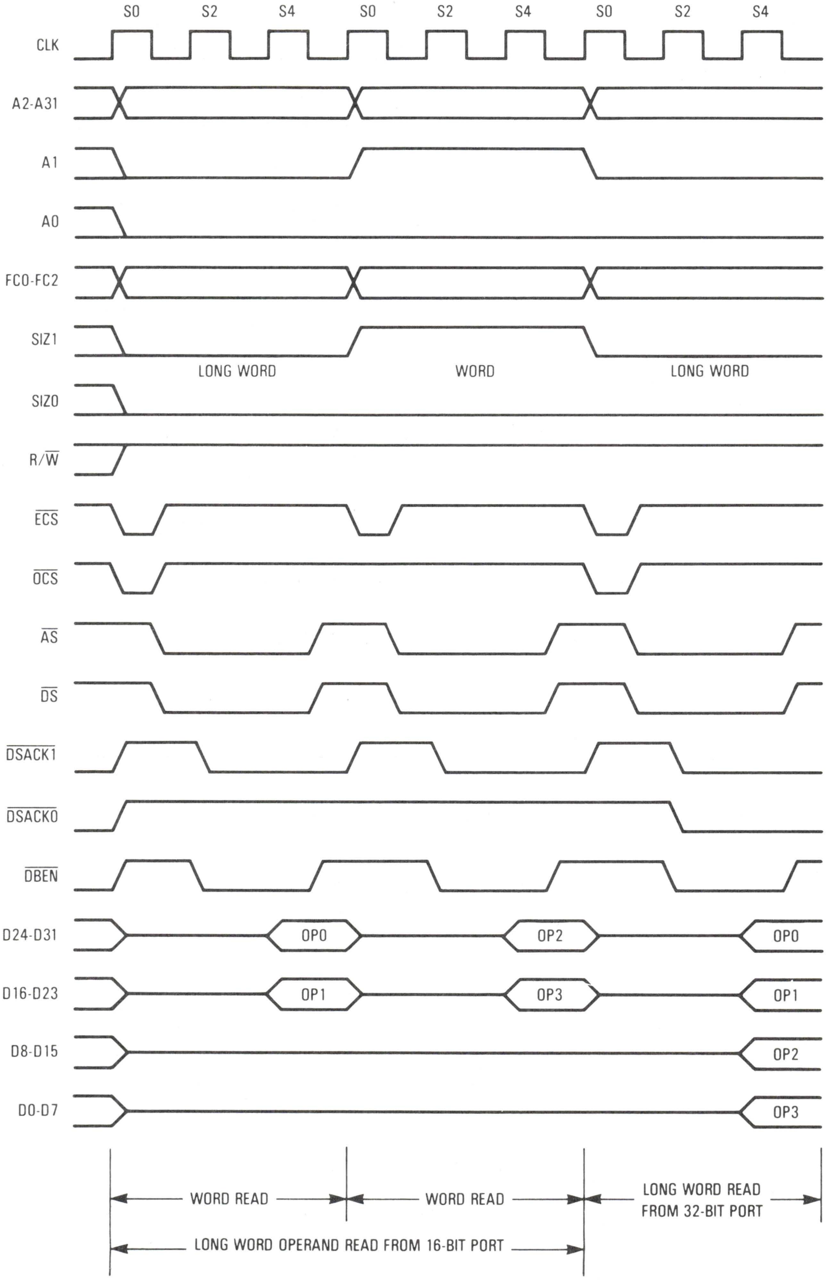

S0
S2
S4
S0
S2
S4
S0
S2
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
LONG WORD
WORD
LONG WORD
SIZ0
R/W̅
E̅C̅S̅
O̅C̅S̅
A̅S̅
D̅S̅
D̅S̅A̅C̅K̅1̅
D̅S̅A̅C̅K̅0̅
D̅B̅E̅N̅
D24-D31
OP0
OP2
OP0
D16-D23
OP1
OP3
OP1
D8-D15
OP2
D0-D7
OP3
WORD READ
WORD READ
LONG WORD READ FROM 32-BIT PORT
LONG WORD OPERAND READ FROM 16-BIT PORT

Figure 7-23. Long-Word Read — 16-Bit Port and 32-Bit Port

State 4

At the end of S4, the processor latches the incoming data.

State 5

The processor negates AS, DS, and DBEN during state 5 (S5). It holds the address valid during S5 to provide address hold time for memory systems. R/W, SIZ1 and SIZ0, and FC2–FC0 also remain valid throughout S5.

The external device keeps its data and DSACKx signals asserted until it detects the negation of AS or DS (whichever it detects first). The device must remove its data and negate DSACKx within approximately one clock period after sensing the negation of AS or DS. DSACKx signals that remain asserted beyond this limit may be prematurely detected for the next bus cycle.

7.3.2 Write Cycle

During a write cycle, the processor transfers data to memory or a peripheral device.

Figure 7-24 is a flowchart of a write cycle operation for a long-word transfer. The following figures show the functional write cycle timing diagrams specified in terms of clock periods. Figure 7-25 shows two write cycles (between two read cycles, with no idle time in between) for a 32-bit port. Figure 7-26 shows byte and word write cycles to a 32-bit port. Figure 7-27 shows a long word write cycle to an 8-bit port. Figure 7-28 shows a long-word write cycle to a 16-bit port.

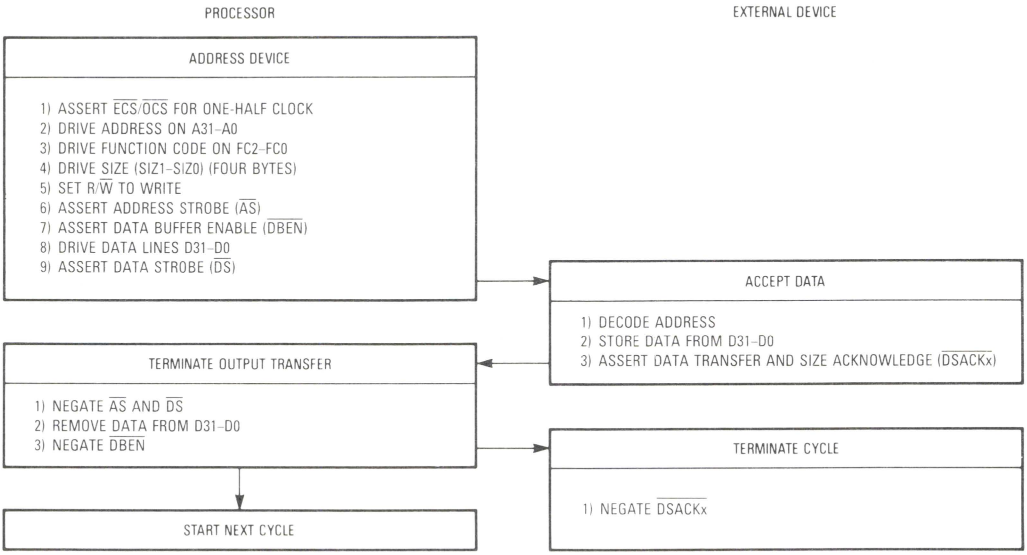

PROCESSOR
EXTERNAL DEVICE
ADDRESS DEVICE
1) ASSERT ECS/OCS FOR ONE-HALF CLOCK
2) DRIVE ADDRESS ON A31–A0
3) DRIVE FUNCTION CODE ON FC2–FC0
4) DRIVE SIZE (SIZ1–SIZ0) (FOUR BYTES)
5) SET R/W TO WRITE
6) ASSERT ADDRESS STROBE (AS)
7) ASSERT DATA BUFFER ENABLE (DBEN)
8) DRIVE DATA LINES D31–D0
9) ASSERT DATA STROBE (DS)
ACCEPT DATA
1) DECODE ADDRESS
2) STORE DATA FROM D31–D0
3) ASSERT DATA TRANSFER AND SIZE ACKNOWLEDGE (DSACKx)
TERMINATE OUTPUT TRANSFER
1) NEGATE AS AND DS
2) REMOVE DATA FROM D31–D0
3) NEGATE DBEN
TERMINATE CYCLE
1) NEGATE DSACKx
START NEXT CYCLE

Figure 7-24. Write Cycle Flowchart

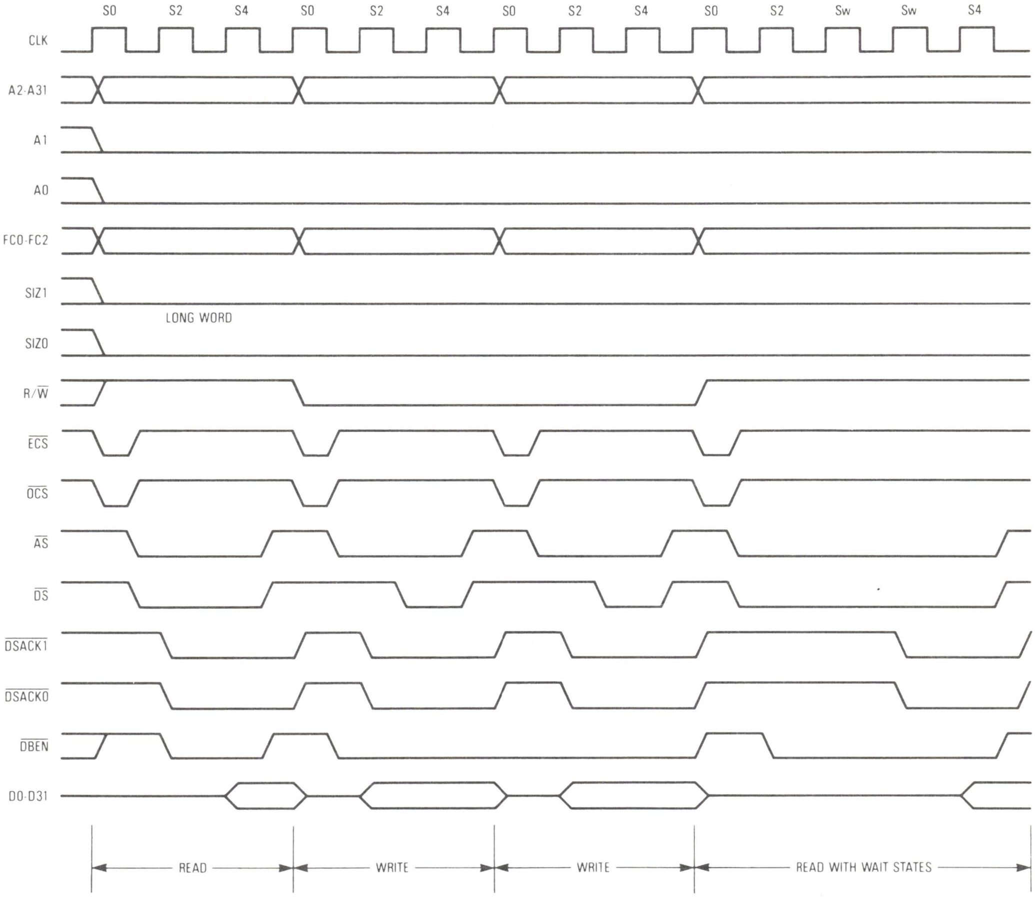

S0
S2
S4
S0
S2
S4
S0
S2
S4
S0
S2
Sw
Sw
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
LONG WORD
SIZ0
R/W̄
ECS̄
OCS̄
AS̄
DS̄
DSACK1̄
DSACK0̄
DBEN̄
D0-D31
READ
WRITE
WRITE
READ WITH WAIT STATES

Figure 7-25. Read-Write-Read Cycles — 32-Bit Port

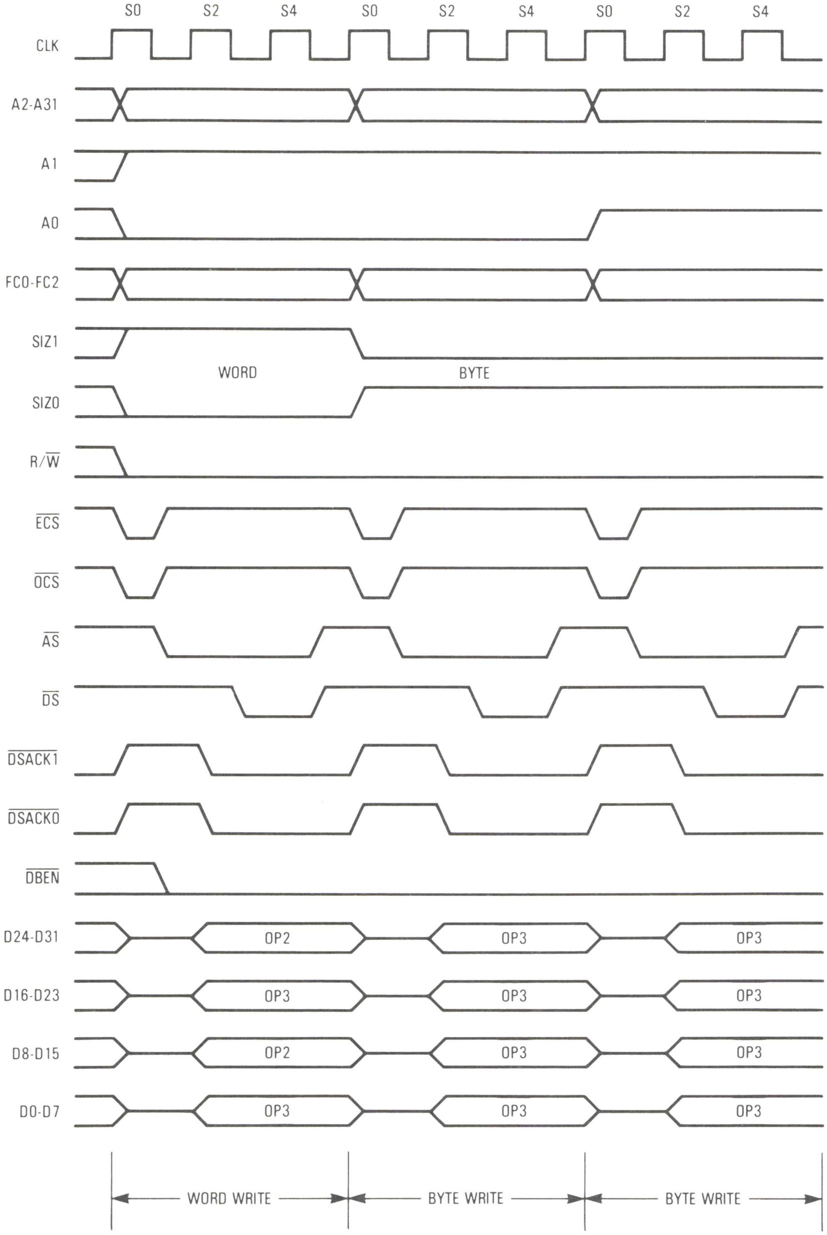

S0
S2
S4
S0
S2
S4
S0
S2
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
WORD
BYTE
SIZ0
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D24-D31
OP2
OP3
OP3
D16-D23
OP3
OP3
OP3
D8-D15
OP2
OP3
OP3
D0-D7
OP3
OP3
OP3
WORD WRITE
BYTE WRITE
BYTE WRITE

Figure 7-26. Byte and Word-Write Cycles — 32-Bit Port

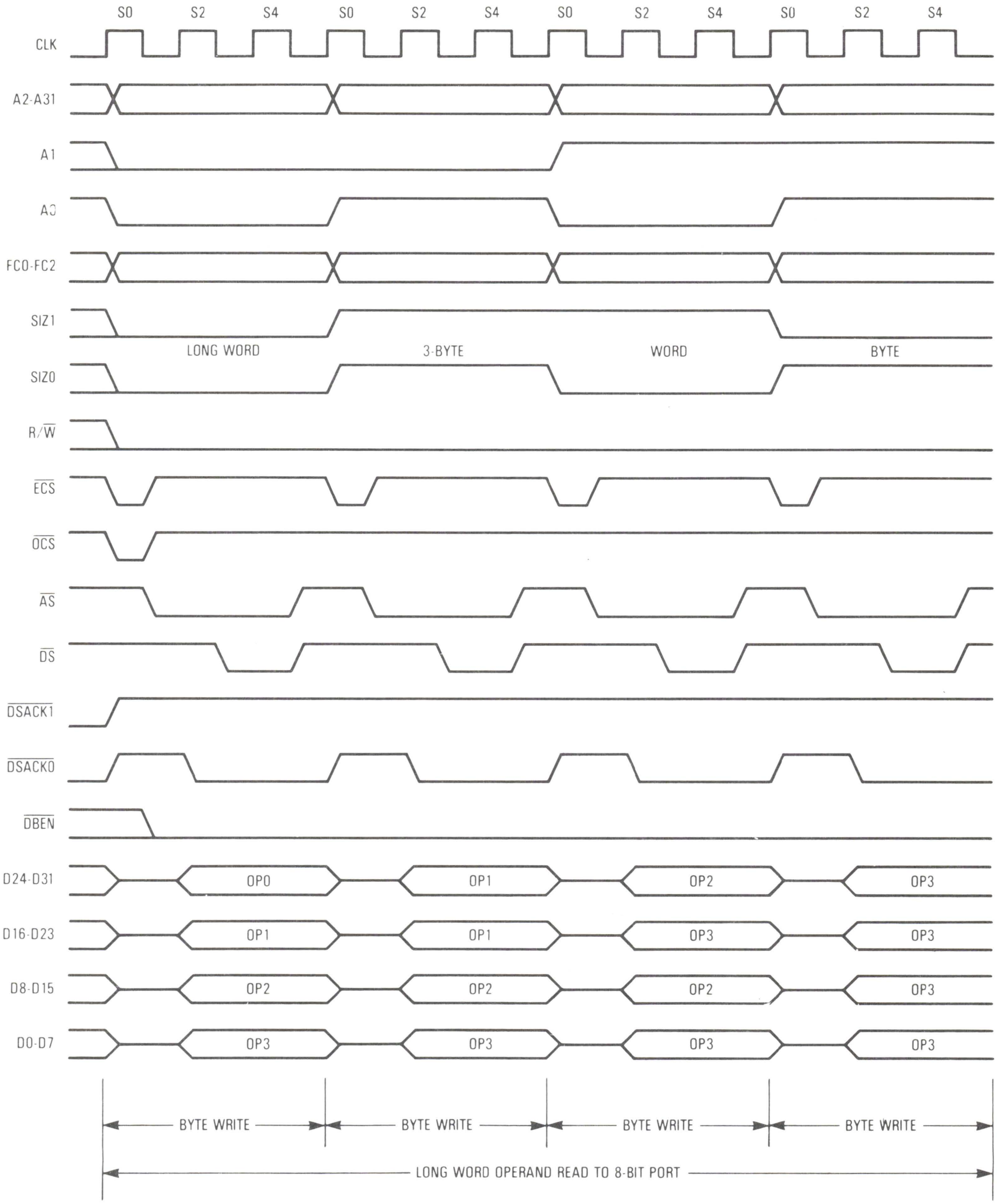

S0
S2
S4
S0
S2
S4
S0
S2
S4
S0
S2
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
LONG WORD
3-BYTE
WORD
BYTE
SIZ0
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D24-D31
OP0
OP1
OP2
OP3
D16-D23
OP1
OP1
OP3
OP3
D8-D15
OP2
OP2
OP2
OP3
D0-D7
OP3
OP3
OP3
OP3
BYTE WRITE
BYTE WRITE
BYTE WRITE
BYTE WRITE
LONG WORD OPERAND READ TO 8-BIT PORT

Figure 7-27. Long-Word Operand Write — 8-Bit Port

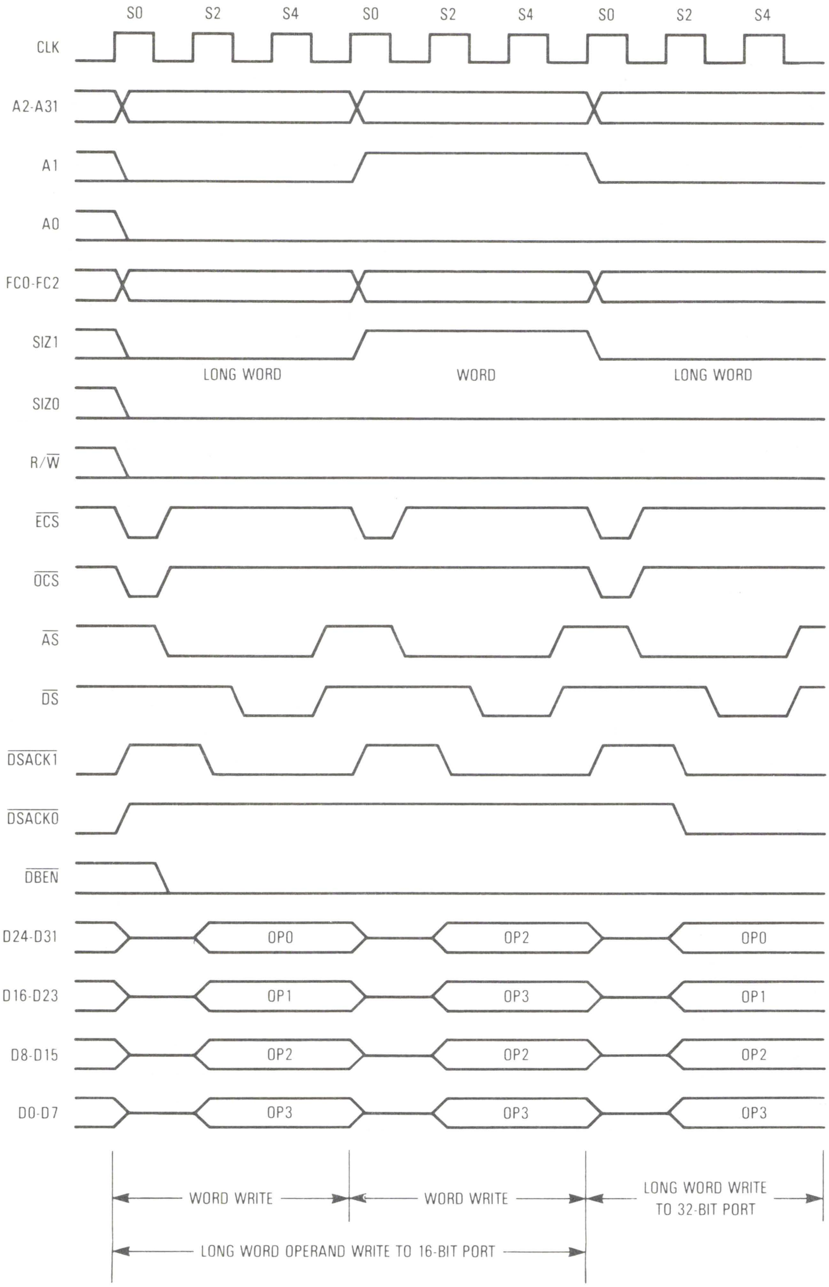

S0
S2
S4
S0
S2
S4
S0
S2
S4
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
LONG WORD
WORD
LONG WORD
SIZ0
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D24-D31
OP0
OP2
OP0
D16-D23
OP1
OP3
OP1
D8-D15
OP2
OP2
OP2
D0-D7
OP3
OP3
OP3
WORD WRITE
WORD WRITE
LONG WORD WRITE TO 32-BIT PORT
LONG WORD OPERAND WRITE TO 16-BIT PORT

Figure 7-28. Long-Word Operand Write — 16-Bit Port

State 0

The write cycle starts in state 0 (S0). The processor drives external cycle start ($\overline{\text{ECS}}$) low, indicating the beginning of an external cycle. When the cycle is the first external cycle of a write operation, operand cycle start ($\overline{\text{OCS}}$) is driven low at the same time. During S0, the processor places a valid address on the address bus (A31–A0) and valid function codes on FC2–FC0. The function codes select the address space for the cycle. The processor drives read/write (R/$\overline{\text{W}}$) low for a write cycle. Size signals SIZ1 and SIZ0 become valid, indicating the number of bytes to be transferred.

State 1

One-half clock later, in state 1 (S1), the processor asserts the address strobe ($\overline{\text{AS}}$) indicating that the address on the address bus is valid. The processor also asserts data buffer enable ($\overline{\text{DBEN}}$) during S1, which can enable external data buffers. In addition, the $\overline{\text{ECS}}$ (and $\overline{\text{OCS}}$, if asserted) signal is negated during S1.

State 2

During state 2 (S2), the processor places the data to be written onto the data bus (D31–D0), and samples the data transfer and size acknowledge signals $\overline{\text{DSACKx}}$ at the end of S2.

State 3

The processor asserts data strobe ($\overline{\text{DS}}$) during state 3 (S3). This indicates that the data is stable on the data bus. As long as at least one of the $\overline{\text{DSACKx}}$ signals is recognized by the end of S2 (meeting the asynchronous input setup time requirement), the cycle terminates one clock later. If $\overline{\text{DSACKx}}$ is not recognized by the start of state 3 (S3), the processor inserts wait states instead of proceeding to states 4 and 5. In order to ensure that wait states are inserted, both $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ must remain negated throughout the asynchronous input setup and hold times around the end of S2. If wait states are added, the processor continues to sample the $\overline{\text{DSACKx}}$ signals on the falling edges of the clock until one is recognized. The selected device uses R/$\overline{\text{W}}$, $\overline{\text{DS}}$, SIZ1–SIZ0, and A1–A0 to latch data from the appropriate byte(s) of the data bus (D31–D24, D23–D16, D15–D8, and D7–D0). The size signals and A1–A0 select the bytes of the data bus. If it has not already done so, the device asserts $\overline{\text{DSACKx}}$ to signal that it has successfully stored the data.

State 4

The processor issues no new control signals during state 4 (S4).

State 5

The processor negates $\overline{\text{AS}}$ and $\overline{\text{DS}}$ during state 5 (S5). It holds the address and data valid during S5 to provide address hold time for memory systems. R/$\overline{\text{W}}$, SIZ1 and SIZ0, FC2–FC0, and $\overline{\text{DBEN}}$ also remain valid throughout S5.

The external device must keep $\overline{\text{DSACKx}}$ asserted until it detects the negation of $\overline{\text{AS}}$ or $\overline{\text{DS}}$ (whichever it detects first). The device must negate $\overline{\text{DSACKx}}$ within approximately one-clock period after sensing the negation of $\overline{\text{AS}}$ or $\overline{\text{DS}}$. $\overline{\text{DSACKx}}$ signals that remain asserted beyond this limit may be prematurely detected for the next bus cycle.

7.3.3 Read-Modify-Write Cycle

The read-modify-write cycle performs a read, conditionally modifies the data in the arithmetic logic unit, and may write the data out to memory. In the MC68020 processor, this operation is indivisible, providing semaphore capabilities for multi-processor systems. During the entire read-modify-write sequence the MC68020 asserts the $\overline{\text{RMC}}$ signal to indicate that an indivisible operation is occurring. The MC68020 does not issue a bus grant ($\overline{\text{BG}}$) signal in response to a bus request ($\overline{\text{BR}}$) signal during this operation.

The test and set (TAS) and compare and swap (CAS and CAS2) instructions are the only MC68020 instructions that utilize read-modify-write operations. Depending on the compare results of the CAS and CAS2 instructions, the write cycle(s) may not occur.

Figure 7-29 is a flowchart of the read-modify-write cycle operation. Figure 7-30 is an example of a functional timing diagram of a TAS instruction specified in terms of clock periods.

State 0

The processor asserts external cycle start ($\overline{\text{ECS}}$) and operand cycle start ($\overline{\text{OCS}}$) in state 0 (S0) to indicate the beginning of an external cycle. The processor also asserts $\overline{\text{RMC}}$ in S0 to identify a read-modify-write cycle. The processor places a valid address on the address bus (A31–A0) and valid function codes on FC2–FC0. The function codes select the address space for the operation. Size signals SIZ1–SIZ0 become valid in S0 to indicate the operand size. The processor drives the read/write signal R/$\overline{\text{W}}$ high for the read cycle.

State 1

One-half clock later, in state 1 (S1), the processor asserts address strobe ($\overline{\text{AS}}$) indicating that the address on the address bus is valid. The processor asserts data strobe ($\overline{\text{DS}}$) also during S1. In addition, the $\overline{\text{ECS}}$ (and $\overline{\text{OCS}}$, if asserted) signal is negated during S1.

State 2

During state 2 (S2), the processor drives $\overline{\text{DBEN}}$ active to enable external data buffers.

The selected device uses R/$\overline{\text{W}}$, SIZ1–SIZ0, A1–A0, and $\overline{\text{DS}}$ to place information on the data bus. Any or all of the bytes (D31–D24, D23–D16, D15–D8, and D7–D0) are selected by the size signals and A1–A0. Concurrently, the selected device may assert the data transfer and size acknowledge ($\overline{\text{DSACKx}}$) signals.

State 3

As long as at least one of the $\overline{\text{DSACKx}}$ signals is recognized by the end of S2 (meeting the asynchronous input setup time requirement), data is latched on the next falling edge of the clock, and the cycle terminates. If $\overline{\text{DSACKx}}$ is not recognized by the start of state 3 (S3), the processor inserts wait states instead of proceeding to states 4 and 5. In order to ensure that wait states are inserted, both $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ must remain negated throughout the asynchronous input setup and hold times around the end of S2. If wait states are added, the processor continues to sample the $\overline{\text{DSACKx}}$ signals on the falling edges of the clock until one is recognized.

State 4

At the end of S4, the processor latches the incoming data.

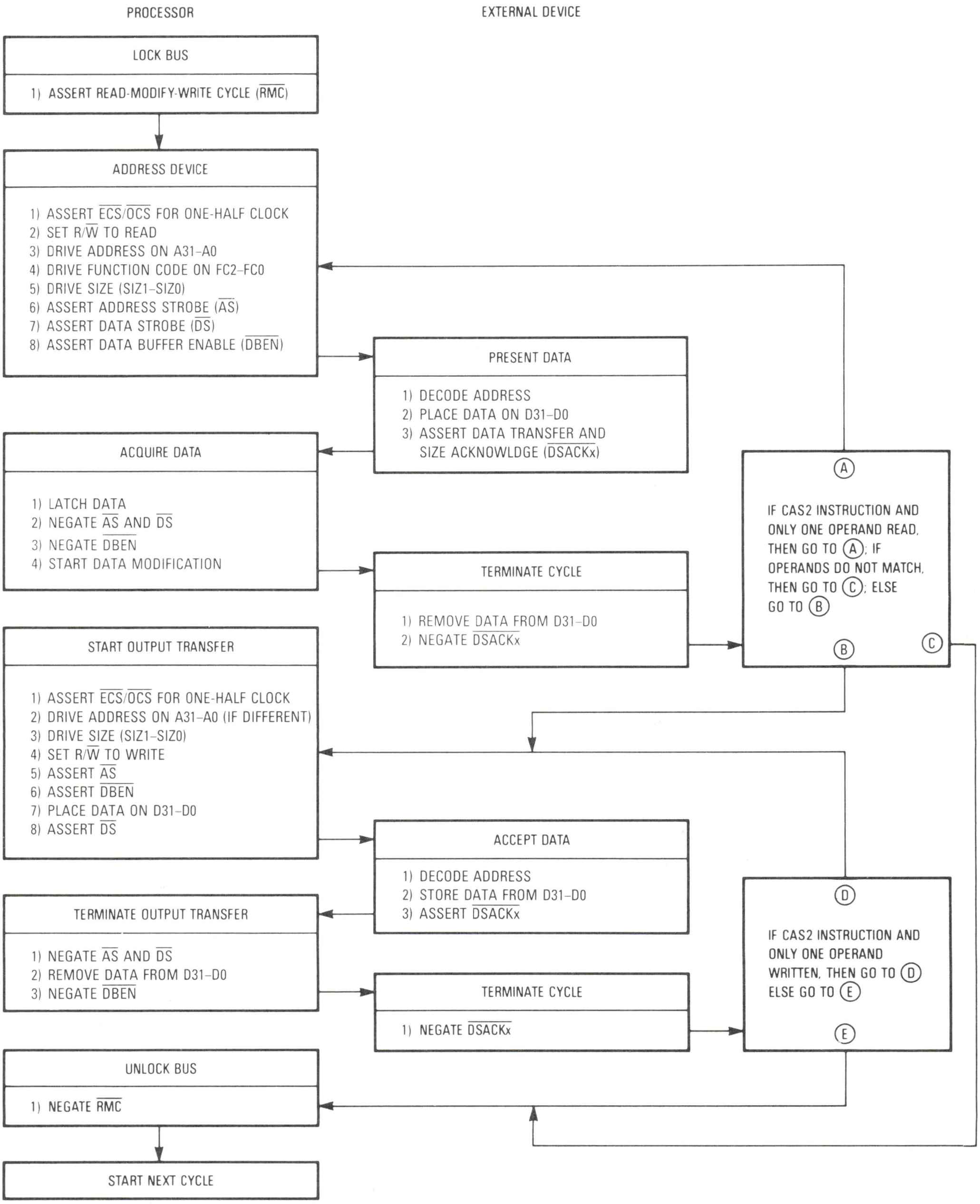

PROCESSOR
EXTERNAL DEVICE
LOCK BUS
1) ASSERT READ-MODIFY-WRITE CYCLE (RMC)
ADDRESS DEVICE
1) ASSERT ECS/OCS FOR ONE-HALF CLOCK
2) SET R/W TO READ
3) DRIVE ADDRESS ON A31–A0
4) DRIVE FUNCTION CODE ON FC2–FC0
5) DRIVE SIZE (SIZ1–SIZ0)
6) ASSERT ADDRESS STROBE (AS)
7) ASSERT DATA STROBE (DS)
8) ASSERT DATA BUFFER ENABLE (DBEN)
PRESENT DATA
1) DECODE ADDRESS
2) PLACE DATA ON D31–D0
3) ASSERT DATA TRANSFER AND SIZE ACKNOWLDGE (DSACKx)
ACQUIRE DATA
1) LATCH DATA
2) NEGATE AS AND DS
3) NEGATE DBEN
4) START DATA MODIFICATION
TERMINATE CYCLE
1) REMOVE DATA FROM D31–D0
2) NEGATE DSACKx
Ⓐ
IF CAS2 INSTRUCTION AND ONLY ONE OPERAND READ, THEN GO TO Ⓐ; IF OPERANDS DO NOT MATCH, THEN GO TO Ⓒ; ELSE GO TO Ⓑ
Ⓑ
Ⓒ
START OUTPUT TRANSFER
1) ASSERT ECS/OCS FOR ONE-HALF CLOCK
2) DRIVE ADDRESS ON A31–A0 (IF DIFFERENT)
3) DRIVE SIZE (SIZ1–SIZ0)
4) SET R/W TO WRITE
5) ASSERT AS
6) ASSERT DBEN
7) PLACE DATA ON D31–D0
8) ASSERT DS
ACCEPT DATA
1) DECODE ADDRESS
2) STORE DATA FROM D31–D0
3) ASSERT DSACKx
Ⓓ
IF CAS2 INSTRUCTION AND ONLY ONE OPERAND WRITTEN, THEN GO TO Ⓓ ELSE GO TO Ⓔ
Ⓔ
TERMINATE OUTPUT TRANSFER
1) NEGATE AS AND DS
2) REMOVE DATA FROM D31–D0
3) NEGATE DBEN
TERMINATE CYCLE
1) NEGATE DSACKx
UNLOCK BUS
1) NEGATE RMC
START NEXT CYCLE

Figure 7-29. Read-Modify-Write Cycle Flowchart

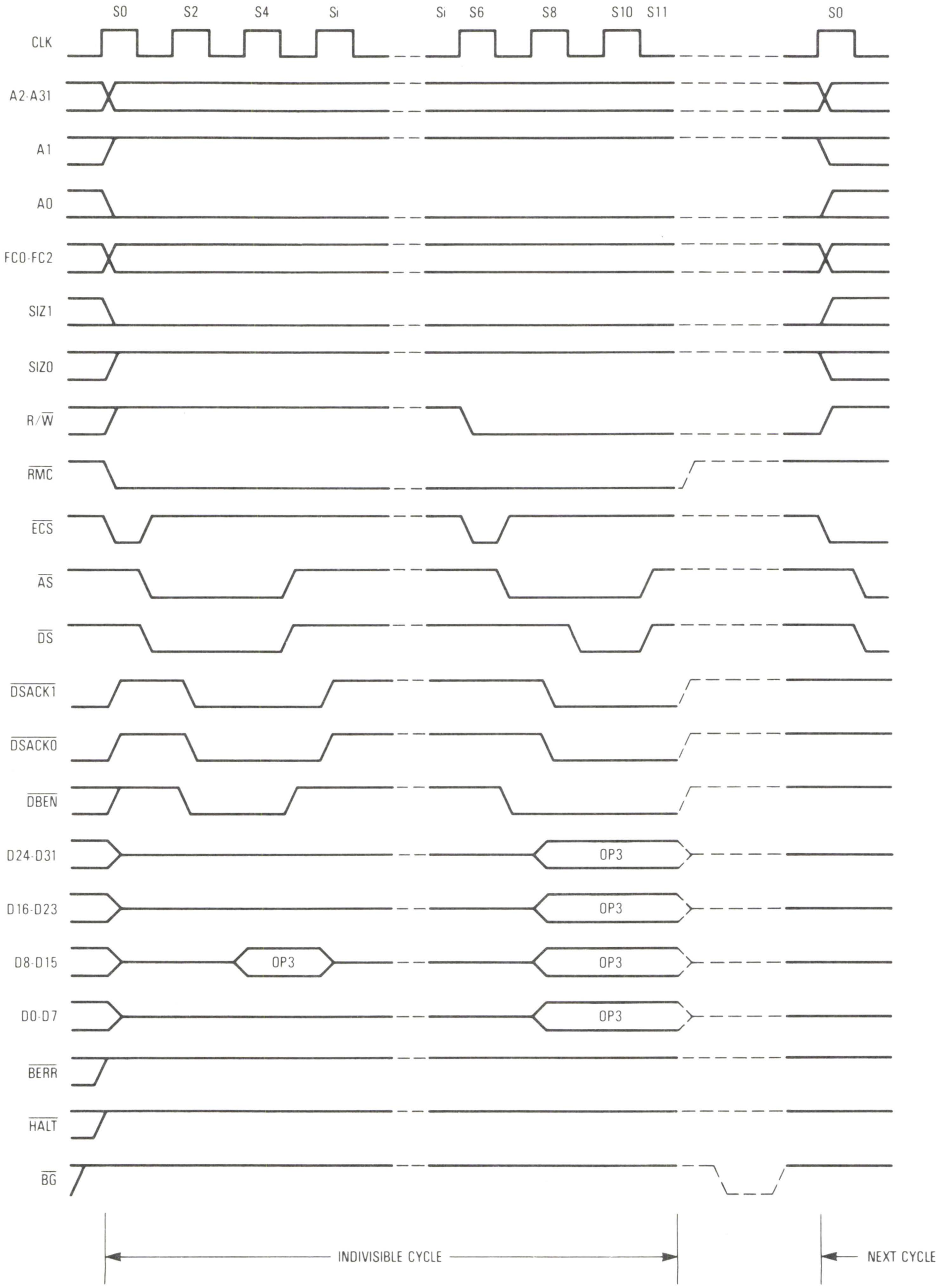

S0
S2
S4
Si
Si
S6
S8
S10
S11
S0
CLK
A2-A31
A1
A0
FC0-FC2
SIZ1
SIZ0
R/$\overline{W}$
$\overline{RMC}$
$\overline{ECS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D24-D31
D16-D23
D8-D15
D0-D7
$\overline{BERR}$
$\overline{HALT}$
$\overline{BG}$
OP3
INDIVISIBLE CYCLE
NEXT CYCLE

Figure 7-30. Byte Read-Modify-Write Cycle — 32-Bit Port (TAS Instruction)

State 5

The processor negates $\overline{\mathrm{AS}}$, $\overline{\mathrm{DS}}$, and $\overline{\mathrm{DBEN}}$ during state 5 (S5). If more than one read cycle is required to read in the operand(s), states S0 through S5 are repeated for each read cycle. When finished reading, the processor holds the address, R/$\overline{\mathrm{W}}$, and FC2–FC0 valid in preparation for the write portion of the cycle.

The external device keeps its data and $\overline{\mathrm{DSACKx}}$ signals asserted until it detects the negation of $\overline{\mathrm{AS}}$ or $\overline{\mathrm{DS}}$ (whichever it detects first). The device must remove the data and negate $\overline{\mathrm{DSACKx}}$ within approximately one clock period after sensing the negation of $\overline{\mathrm{AS}}$ or $\overline{\mathrm{DS}}$. $\overline{\mathrm{DSACKx}}$ signals that remain asserted beyond this limit may be prematurely detected for the next portion of the operation.

Idle States

The processor does not assert any new control signals during the idle states, but it may internally begin the modify portion of the cycle at this time. States 6 through 11 are omitted if no write cycle is required. If a write cycle is required, the R/$\overline{\mathrm{W}}$ signal remains in the read mode until state 6 to prevent bus conflicts with the preceding read portion of the cycle; the data bus is not driven until state 8.

State 6

The processor asserts $\overline{\mathrm{ECS}}$ and $\overline{\mathrm{OCS}}$ in state 6 (S6) to indicate that another external cycle is beginning. The processor drives R/$\overline{\mathrm{W}}$ low for a write cycle. Depending on the write operation to be performed, the address lines may change during state 6.

State 7

In state 7 (S7), the processor asserts address strobe ($\overline{\mathrm{AS}}$) indicating that the address on the address bus is valid. The processor also asserts data buffer enable ($\overline{\mathrm{DBEN}}$), which can be used to enable data buffers, during S7. In addition, the $\overline{\mathrm{ECS}}$ (and $\overline{\mathrm{OCS}}$,if asserted) signal is negated during S7.

State 8

During state 8 (S8) the processor places the data to be written onto the data bus (D31–D0).

State 9

The processor asserts the data strobe ($\overline{\mathrm{DS}}$) during state 9 (S9). This indicates that the data is stable on the data bus. As long as at least one of the $\overline{\mathrm{DSACKx}}$ signals is recognized by the end of S8 (meeting the asynchronous input setup time requirement), the cycle terminates one clock later. If $\overline{\mathrm{DSACKx}}$ is not recognized by the start of S9, the processor inserts wait states instead of proceeding to states 10 and 11. In order to ensure that wait states are inserted, both $\overline{\mathrm{DSACK1}}$ and $\overline{\mathrm{DSACK0}}$ must remain negated throughout the asynchronous input setup and hold times around the end of S8. If wait states are added, the processor continues to sample $\overline{\mathrm{DSACKx}}$ signals on the falling edges of the clock until one is recognized.

The selected device uses R/$\overline{\mathrm{W}}$, $\overline{\mathrm{DS}}$, SIZ1–SIZ0, and A1–A0 to latch data from the appropriate section(s) of the data bus (D31–D24, D23–D16, D15–D8, and D7–D0). The size signals and A1–A0 select the data bus sections. If it has not already done so, the device asserts $\overline{\mathrm{DSACKx}}$ when it has successfully stored the data.

State 10

The processor issues no new control signals during state 10 (S10).

State 11

The processor negates $\overline{\mathrm{AS}}$ and $\overline{\mathrm{DS}}$ during state 11 (S11). It holds the address and data valid during S11 to provide address hold time for memory systems. R/$\overline{\mathrm{W}}$ and FC2–FC0 also remain valid throughout S11.

If more than one write cycle is required, states S6 through S11 are repeated for each write cycle.

The external device keeps $\overline{\mathrm{DSACKx}}$ asserted until it detects the negation of $\overline{\mathrm{AS}}$ or $\overline{\mathrm{DS}}$ (whichever it detects first). The device must remove its data and negate $\overline{\mathrm{DSACKx}}$ within approximately one-clock period after sensing the negation of $\overline{\mathrm{AS}}$ or $\overline{\mathrm{DS}}$.

7.4 CPU SPACE CYCLES

The function codes (FC2–FC0) select user and supervisor program and data areas, as listed in Table 4-1. The area selected by function code FC2–FC0 = \$7 is classified as the CPU space. The interrupt acknowledge, breakpoint acknowledge, module operations, and coprocessor communication cycles described in the following paragraphs utilize CPU space.

The CPU space type is encoded on address signals A19–A16 during a CPU space operation and indicates the function that the processor is performing. On the MC68020, four of the encodings are implemented as shown in Figure 7-31. All unused values are reserved by Motorola for future additional CPU space types.

7.4.1 Interrupt Acknowledge Bus Cycles

When a peripheral device signals the processor (with the $\overline{\mathrm{IPL2}}$–$\overline{\mathrm{IPL0}}$ signals) that the device requires service, and the internally synchronized value on these signals indicates a higher

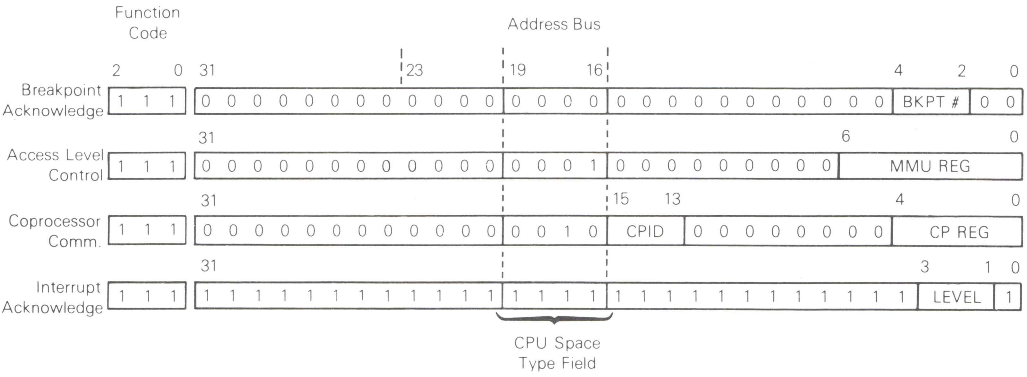

Function Code
Address Bus
2 0 31 23 19 16 4 2 0
Breakpoint Acknowledge
1 1 1 | 0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 | BKPT # | 0 0
31 6 0
Access Level Control
1 1 1 | 0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 0 1 | 0 0 0 0 0 0 0 0 0 | MMU REG
31 15 13 4 0
Coprocessor Comm.
1 1 1 | 0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 1 0 | CPID | 0 0 0 0 0 0 0 0 | CP REG
31 3 1 0
Interrupt Acknowledge
1 1 1 | 1 1 1 1 1 1 1 1 1 1 1 1 | 1 1 1 1 | 1 1 1 1 1 1 1 1 1 1 1 1 | LEVEL | 1
CPU Space Type Field

Figure 7-31. MC68020 CPU-Space Address Encoding

priority than the interrupt mask in the status register (or that a transition has occurred in the case of a level 7 interrupt), the processor makes the interrupt a pending interrupt. Refer to **8.1.9 Interrupt Exceptions** for details on the recognition of interrupts.

The MC68020 takes an interrupt exception for a pending interrupt within one instruction boundary (after processing any other pending exception with a higher priority). The following paragraphs describe the various kinds of interrupt acknowledge bus cycles that can be executed as part of interrupt exception processing.

7.4.1.1 INTERRUPT ACKNOWLEDGE CYCLE — TERMINATED NORMALLY. When the MC68020 processes an interrupt exception, it performs an interrupt acknowledge cycle to obtain the number of the vector that contains the starting location of the interrupt service routine.

Some interrupting devices have programmable vector registers that contain the interrupt vectors for the routines they use. The following paragraphs describe the interrupt acknowledge cycle for these devices. Other interrupting conditions or devices cannot supply a vector number, and use the autovector cycle described in **7.4.1.2 AUTOVECTOR INTERRUPT ACKNOWLEDGE CYCLE**.

The interrupt acknowledge cycle is a read cycle. It differs from the read cycle described in **7.3.1 Read Cycle** in that it accesses the CPU address space. Specifically, the differences are:

1. The function codes (FC2–FC0) are set to seven (FC2/FC1/FC1 = 111) for CPU address space.
2. Address signals A3, A2, and A1 are set to the interrupt request level (the inverted values of $\overline{\text{IPL2}}$, $\overline{\text{IPL1}}$, and $\overline{\text{IPL0}}$, respectively).
3. The CPU space type field (address signals A19–A16) is set to $F, the interrupt acknowledge code.
4. Other address signals (A31–A20, A15–A4, and A0) are set to 1.

The responding device places the vector number on the data bus during the interrupt acknowledge cycle. Beyond this, the cycle is terminated normally with $\overline{\text{DSACKx}}$. Figure 7-32 is the flowchart of the interrupt acknowledge cycle.

Figure 7-33 shows the timing for an interrupt acknowledge cycle terminated with $\overline{\text{DSACKx}}$.

7.4.1.2 AUTOVECTOR INTERRUPT ACKNOWLEDGE CYCLE. When the interrupting device cannot supply a vector number, it requests an automatically generated vector, or "autovector". Instead of placing a vector number on the data bus and asserting the data transfer and size acknowledge signals ($\overline{\text{DSACKx}}$) the device asserts the autovector signal ($\overline{\text{AVEC}}$) to terminate the cycle. The $\overline{\text{DSACKx}}$ signals may not be asserted during an interrupt acknowledge cycle terminated by $\overline{\text{AVEC}}$.

The vector number supplied in an autovector operation is derived from the interrupt level of the current interrupt. When the $\overline{\text{AVEC}}$ signal is asserted instead of $\overline{\text{DSACK}}$ during an

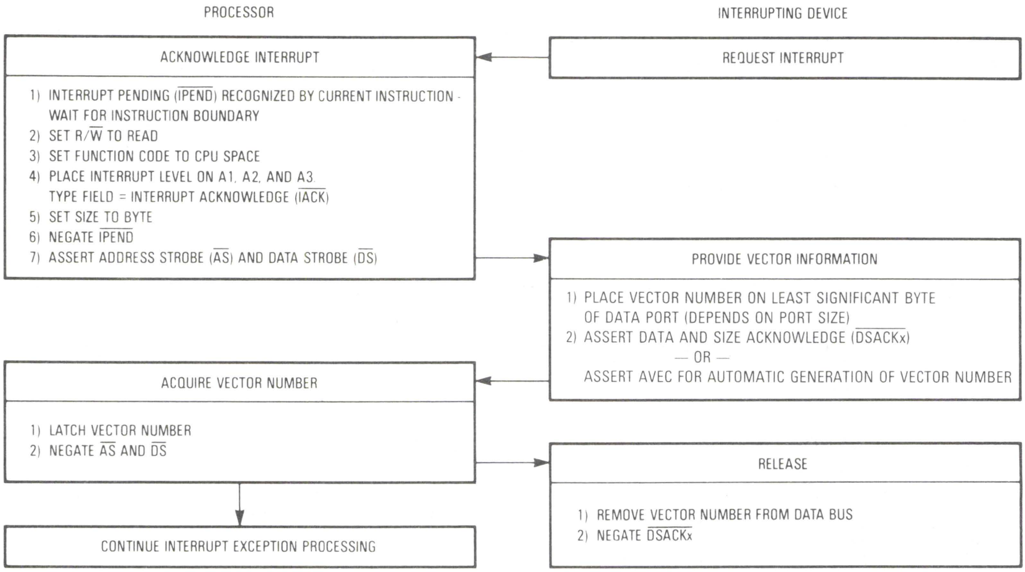

PROCESSOR
INTERRUPTING DEVICE
ACKNOWLEDGE INTERRUPT
REQUEST INTERRUPT
1) INTERRUPT PENDING (IPEND) RECOGNIZED BY CURRENT INSTRUCTION - WAIT FOR INSTRUCTION BOUNDARY
2) SET R/W TO READ
3) SET FUNCTION CODE TO CPU SPACE
4) PLACE INTERRUPT LEVEL ON A1, A2, AND A3 TYPE FIELD = INTERRUPT ACKNOWLEDGE (IACK)
5) SET SIZE TO BYTE
6) NEGATE IPEND
7) ASSERT ADDRESS STROBE (AS) AND DATA STROBE (DS)
PROVIDE VECTOR INFORMATION
1) PLACE VECTOR NUMBER ON LEAST SIGNIFICANT BYTE OF DATA PORT (DEPENDS ON PORT SIZE)
2) ASSERT DATA AND SIZE ACKNOWLEDGE (DSACKx)
— OR —
ASSERT AVEC FOR AUTOMATIC GENERATION OF VECTOR NUMBER
ACQUIRE VECTOR NUMBER
1) LATCH VECTOR NUMBER
2) NEGATE AS AND DS
RELEASE
1) REMOVE VECTOR NUMBER FROM DATA BUS
2) NEGATE DSACKx
CONTINUE INTERRUPT EXCEPTION PROCESSING

Figure 7-32. Interrupt Acknowledge Cycle Flowchart

interrupt acknowledge cycle, the MC68020 ignores the state of the data bus and internally generates the vector number, the sum of the interrupt level plus 24 ($18). There are seven distinct autovectors that can be used, corresponding to the seven levels of interrupt available with signals IPL2–IPL0. Figure 7-34 shows the timing for an autovector operation.

7.4.1.3 SPURIOUS INTERRUPT CYCLE. When a device does not respond to an interrupt acknowledge cycle with AVEC or DSACKx, the external logic typically returns the bus error signal (BERR). The MC68020 automatically generates the spurious interrupt vector number, 24, instead of the interrupt vector number in this case. If the halt signal HALT is also asserted, the processor retries the cycle.

7.4.2 Breakpoint Acknowledge Cycle

The breakpoint acknowledge cycle is generated by the execution of a breakpoint instruction (BKPT). The breakpoint acknowledge cycle allows the external hardware to provide an instruction word directly into the instruction pipeline as the program executes. This cycle accesses the CPU space, with a type field of zero, and provides the breakpoint number specified by the instruction on address lines A4–A2. If the external hardware terminates the cycle with DSACKx, the data on the bus (an instruction word) is inserted into the instruction pipe replacing the breakpoint opcode, and is executed after the breakpoint acknowledge cycle completes. The breakpoint instruction requires a word to be transferred so if the first bus cycle accesses an 8-bit port, a second cycle is required. If the external logic terminates the breakpoint acknowledge cycle with BERR (i.e., no instruction word

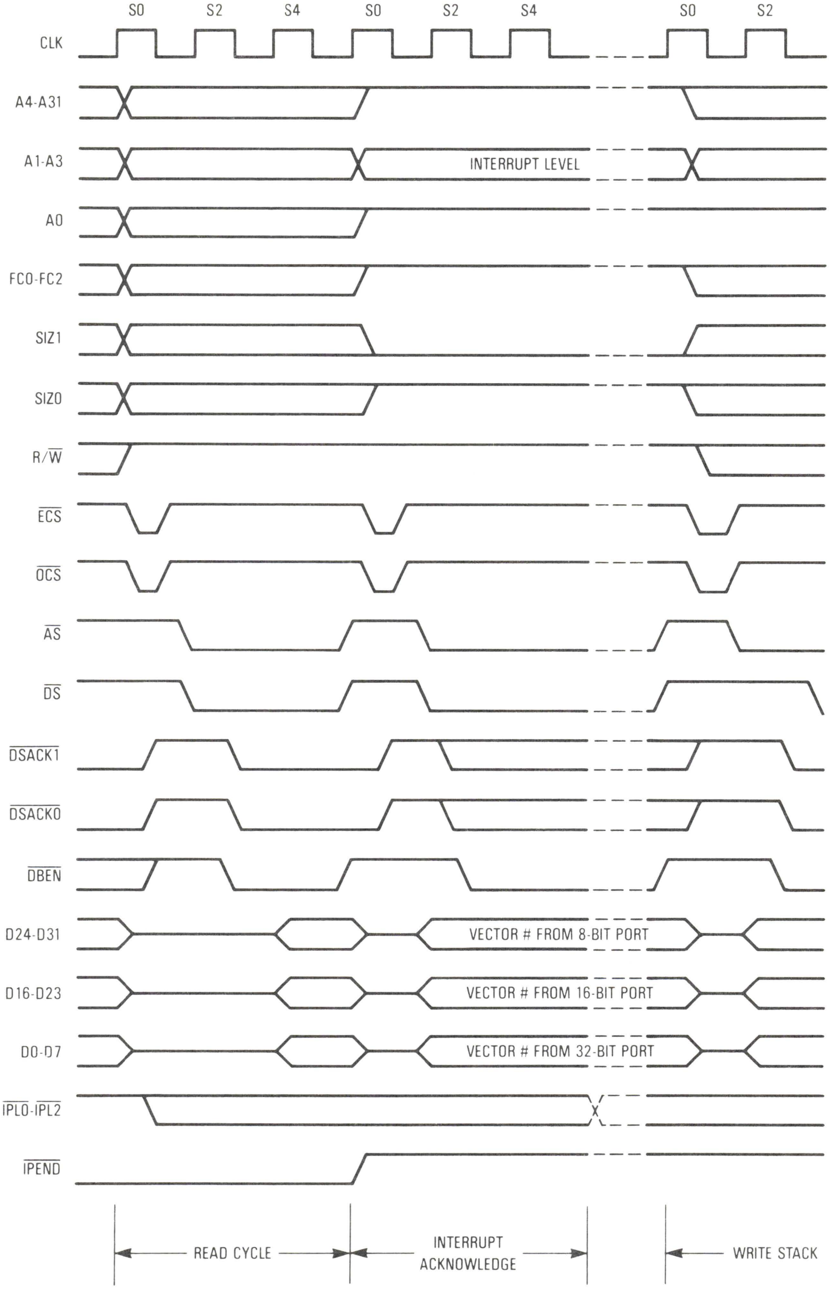

S0
S2
S4
S0
S2
S4
S0
S2
CLK
A4-A31
A1-A3
INTERRUPT LEVEL
A0
FC0-FC2
SIZ1
SIZ0
R/W̄
ECS̄
OCS̄
AS̄
DS̄
DSACK1̄
DSACK0̄
DBEN̄
D24-D31
VECTOR # FROM 8-BIT PORT
D16-D23
VECTOR # FROM 16-BIT PORT
D0-D7
VECTOR # FROM 32-BIT PORT
IPL0-IPL2̄
IPEND̄
READ CYCLE
INTERRUPT ACKNOWLEDGE
WRITE STACK

Figure 7-33. Interrupt Acknowledge Cycle Timing

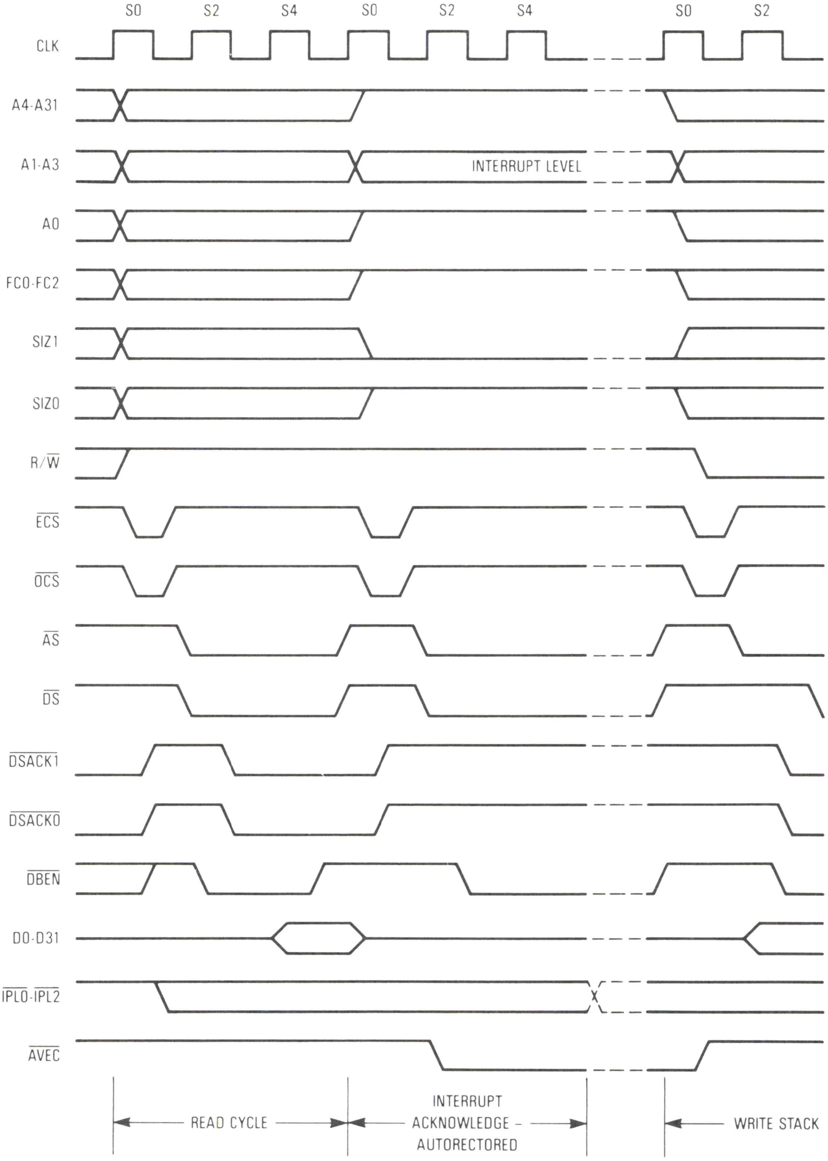

S0
S2
S4
S0
S2
S4
S0
S2
CLK
A4-A31
A1-A3
INTERRUPT LEVEL
A0
FC0-FC2
SIZ1
SIZ0
R/W
ECS
OCS
AS
DS
DSACK1
DSACK0
DBEN
D0-D31
IPL0-IPL2
AVEC
READ CYCLE
INTERRUPT ACKNOWLEDGE - AUTORECTORED
WRITE STACK

Figure 7-34. Autovector Operation Timing

available) the processor takes an illegal instruction exception. Figure 7-35 is a flowchart of the breakpoint acknowledge cycle. Figure 7-36 shows the timing for a breakpoint acknowledge cycle that returns an instruction word. Figure 7-37 shows the timing for a breakpoint acknowledge cycle that signals an exception.

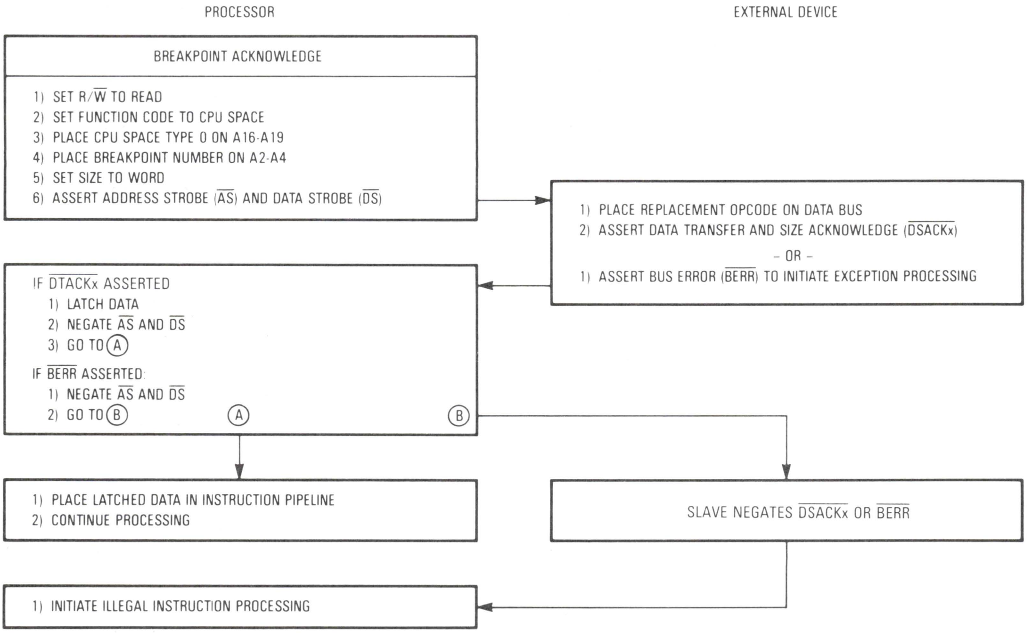

PROCESSOR
EXTERNAL DEVICE
BREAKPOINT ACKNOWLEDGE
1) SET R/$\overline{\mathrm{W}}$ TO READ
2) SET FUNCTION CODE TO CPU SPACE
3) PLACE CPU SPACE TYPE 0 ON A16-A19
4) PLACE BREAKPOINT NUMBER ON A2-A4
5) SET SIZE TO WORD
6) ASSERT ADDRESS STROBE ($\overline{\mathrm{AS}}$) AND DATA STROBE ($\overline{\mathrm{DS}}$)
1) PLACE REPLACEMENT OPCODE ON DATA BUS
2) ASSERT DATA TRANSFER AND SIZE ACKNOWLEDGE ($\overline{\mathrm{DSACKx}}$)
– OR –
1) ASSERT BUS ERROR ($\overline{\mathrm{BERR}}$) TO INITIATE EXCEPTION PROCESSING
IF $\overline{\mathrm{DTACKx}}$ ASSERTED
1) LATCH DATA
2) NEGATE $\overline{\mathrm{AS}}$ AND $\overline{\mathrm{DS}}$
3) GO TO Ⓐ
IF $\overline{\mathrm{BERR}}$ ASSERTED:
1) NEGATE $\overline{\mathrm{AS}}$ AND $\overline{\mathrm{DS}}$
2) GO TO Ⓑ
Ⓐ
Ⓑ
1) PLACE LATCHED DATA IN INSTRUCTION PIPELINE
2) CONTINUE PROCESSING
SLAVE NEGATES $\overline{\mathrm{DSACKx}}$ OR $\overline{\mathrm{BERR}}$
1) INITIATE ILLEGAL INSTRUCTION PROCESSING

Figure 7-35. Breakpoint Operation Flow

7.4.3 Coprocessor Communication Cycles

The MC68020 coprocessor interface provides instruction-oriented communication between the processor and as many as eight coprocessors. The bus communication required to support coprocessor operations uses the MC68020 CPU space, with a type field of $2.

Coprocessor accesses use the MC68020 bus protocol except that the address bus supplies access information rather than a 32-bit address. The CPU space type field (address signals A19–A16) for a coprocessor operation is $2. Address signals A15–A13 contain the coprocessor identification number (Cp-ID), and address signals A5–A0 specify the coprocessor interface register to be accessed. The memory management unit of an MC68020 system is always identified by coprocessor ID zero and has an extended register select field (A7–A0) in CPU space 0001 for use by the CALLM and RTM access level checking mechanism. Refer to **SECTION 11 APPLICATIONS INFORMATION** for more details.

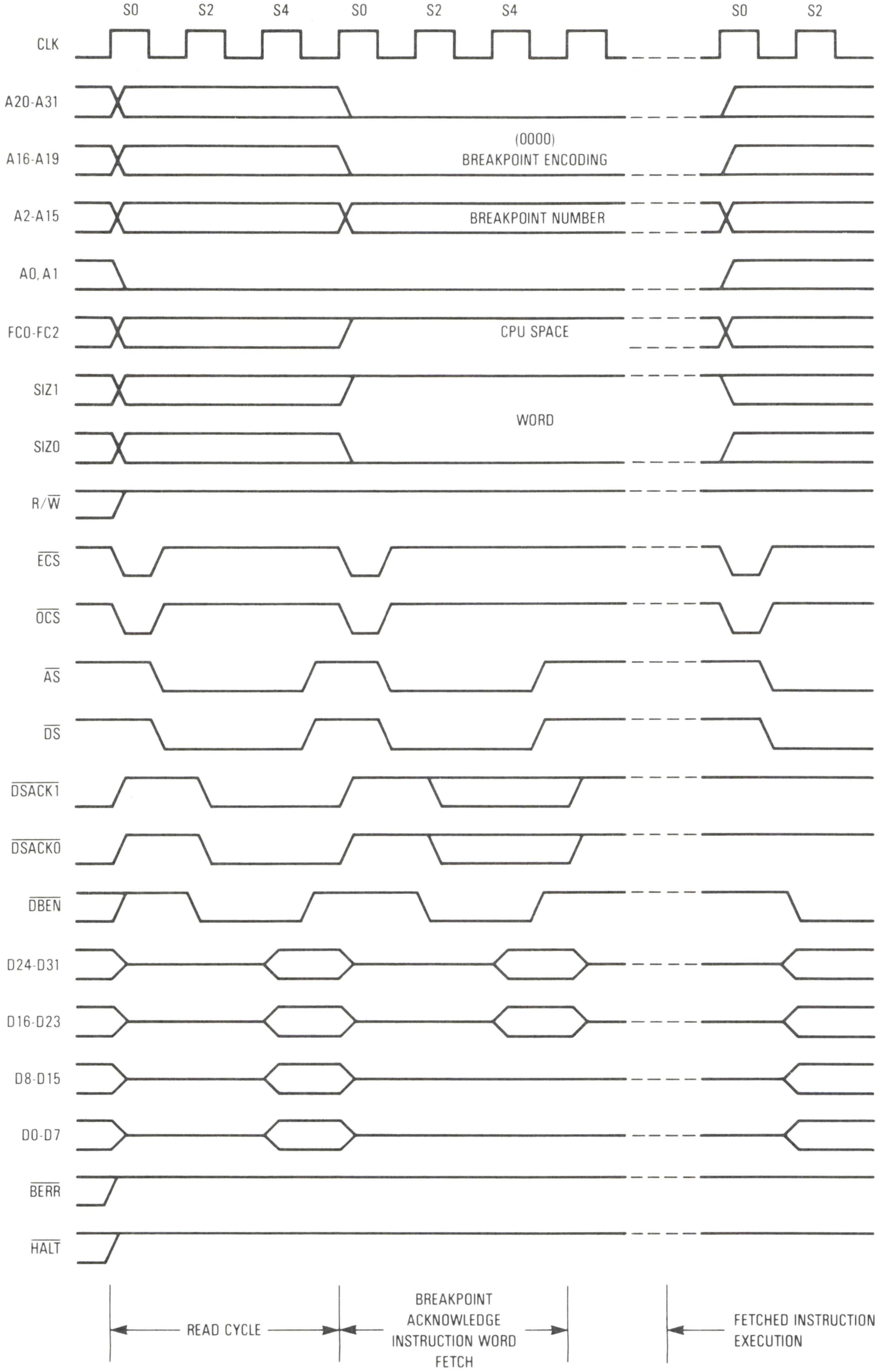

S0
S2
S4
S0
S2
S4
S0
S2
CLK
A20-A31
A16-A19
(0000)
BREAKPOINT ENCODING
A2-A15
BREAKPOINT NUMBER
A0, A1
FC0-FC2
CPU SPACE
SIZ1
WORD
SIZ0
R/W̄
ECS̄
OCS̄
ĀS̄
D̄S̄
DSACK1̄
DSACK0̄
DBEN̄
D24-D31
D16-D23
D8-D15
D0-D7
BERR̄
HALT̄
READ CYCLE
BREAKPOINT ACKNOWLEDGE INSTRUCTION WORD FETCH
FETCHED INSTRUCTION EXECUTION

Figure 7-36. Breakpoint Acknowledge Cycle Timing

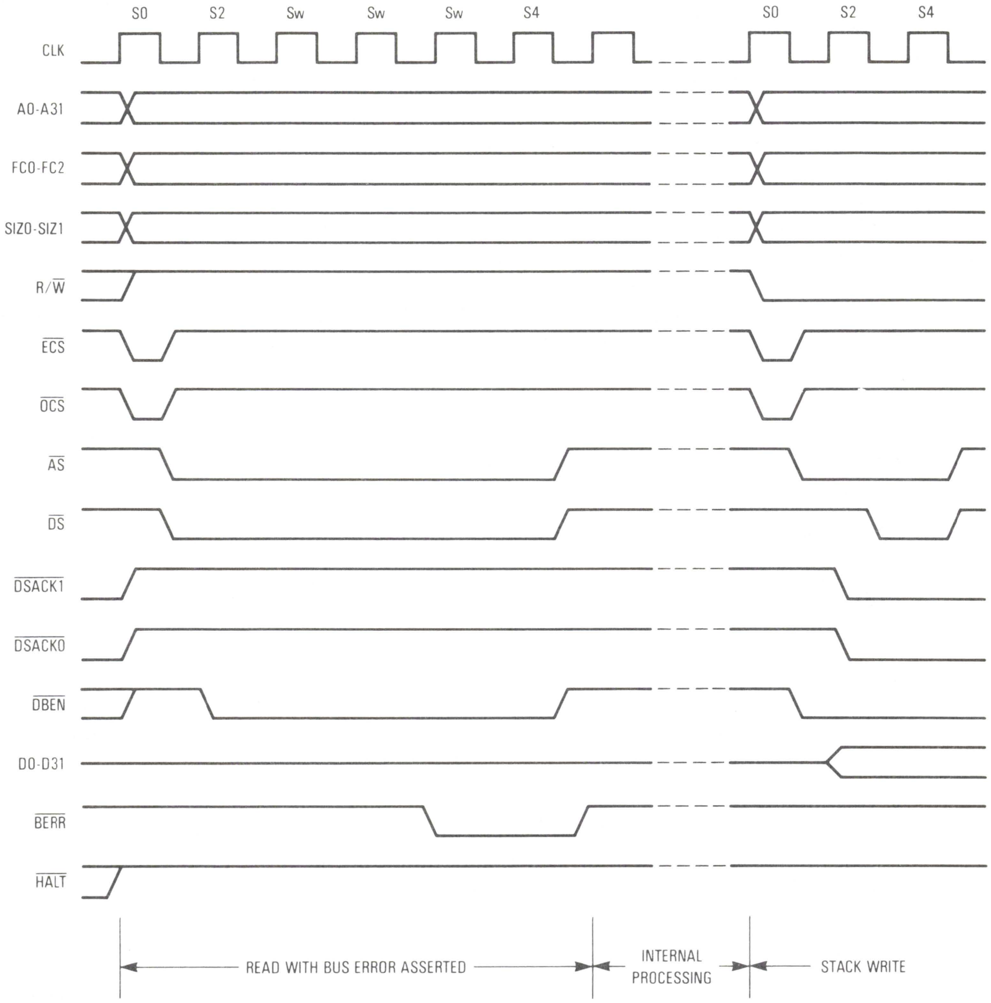

S0
S2
Sw
Sw
Sw
S4
S0
S2
S4
CLK
A0-A31
FC0-FC2
SIZ0-SIZ1
R/W̅
E̅C̅S̅
O̅C̅S̅
A̅S̅
D̅S̅
D̅S̅A̅C̅K̅1̅
D̅S̅A̅C̅K̅0̅
D̅B̅E̅N̅
D0-D31
B̅E̅R̅R̅
H̅A̅L̅T̅
READ WITH BUS ERROR ASSERTED
INTERNAL PROCESSING
STACK WRITE

Figure 7-37. Breakpoint Acknowledge Cycle Timing (Exception Signaled)

7.5 BUS EXCEPTION CONTROL CYCLES

The MC68020 bus architecture requires assertion of DSACKx from an external device to signal that a bus cycle is complete. DSACKx or AVEC is not asserted in these cases:

- The external device does not respond.
- No interrupt vector is provided.
- Various other application dependent errors occur.

External circuitry can provide the bus error signal (BERR) when no device responds by asserting DSACKx or AVEC within an appropriate period of time after the processor asserts the address strobe (AS). This allows the cycle to terminate and the processor to enter exception processing for the error condition.

Another signal that is used for bus exception control is the halt signal (HALT). This signal can be asserted by an external device for debugging purposes to cause single bus cycle operation, or (in combination with BERR) a retry of a bus cycle in error.

In order to properly control termination of a bus cycle for a retry or a bus error condition, DSACKx, BERR, and HALT can be asserted and negated with the rising edge of the MC68020 clock. This assures that when two signals are asserted simultaneously, the required setup time (#47A) and hold time (#47B) for both of them is met for the same falling edge of the processor clock. (Refer to **SECTION 12 ELECTRICAL CHARACTERISTICS** for timing requirements.) This or some equivalent precaution should be designed into the external circuitry that provides these signals.

The acceptable bus cycle terminations for asynchronous cycles are summarized in relation to DSACKx assertion as follows (case numbers refer to Table 7-8).

Normal Termination:

DSACKx is asserted, BERR, and HALT remain negated (case 1).

Halt Termination:

HALT is asserted at same time, or before DSACKx, and BERR remains negated (case 2).

Bus Error Termination:

BERR is asserted in lieu of, at the same time, or before DSACKx (case 3) or after DSACKx (case 4) and HALT remains negated; BERR is negated at the same time or after DSACKx.

Retry Termination:

HALT and BERR are asserted in lieu of, at the same time, or before DSACKx (case 5); or after DSACKx (case 6); BERR is negated at the same time or after DSACKx, HALT may be negated at the same time, or after BERR.

Table 7-8 shows various combinations of control signal sequences and the resulting bus cycle terminations. To ensure predictable operation, BERR and HALT should be negated according to specifications #28 and #57 in **SECTION 12 ELECTRICAL CHARACTERISTICS**. DSACKx, BERR, and HALT may be negated after AS. If DSACKx or BERR remain asserted into S2 of the next bus cycle, that cycle may be terminated prematurely.

EXAMPLE A:

A system uses a watch-dog timer to terminate accesses to an unpopulated address space. The timer asserts BERR after time out (case 3).

EXAMPLE B:

A system uses error detection and correction on RAM contents. The designer may:

1. Delay DSACKx until data is verified, and assert BERR and HALT simultaneously to indicate to the processor to automatically retry the error cycle (case 5), or if data is valid assert DSACKx (case 1).

Table 7-8. DSACK, BERR, and HALT Assertion Results

Case No.	Control Signal	Asserted on Rising Edge of State		Result
		N	N+2	
1	DSACKx	A	S	Normal cycle terminate and continue.
	BERR	NA	NA	
	HALT	NA	X	
2	DSACKx	A	S	Normal cycle terminate and halt. Continue when HALT negated.
	BERR	NA	NA	
	HALT	A/S	S	
3	DSACKx	NA/A	X	Terminate and take bus error exception, possibly deferred.
	BERR	A	S	
	HALT	NA	NA	
4	DSACKx	A	X	Terminate and take bus error exception, possibly deferred.
	BERR	NA	A	
	HALT	NA	NA	
5	DSACKx	NA/A	X	Terminate and retry when HALT negated.
	BERR	A	S	
	HALT	A/S	S	
6	DSACKx	A	X	Terminate and retry when HALT negated.
	BERR	NA	A	
	HALT	NA	A	

LEGEND:
N — The number of current even bus state (e.g., S2, S4, etc.)
A — Signal is asserted in this bus state
NA — Signal is not asserted in this state
X — Don't care
S — Signal was asserted in previous state and remains asserted in this state

2. Delay DSACKx until data is verified, and assert BERR with or without DSACKx if data is in error (case 3). This initiates exception processing for software handling of the condition.
3. Return DSACKx prior to data verification. If data is invalid, BERR is asserted on the next clock cycle (case 4). This initiates exception processing for software handling of the condition.
4. Return DSACKx prior to data verification; if data is invalid, assert BERR and HALT on the next clock cycle (case 6). The memory controller can then correct the RAM prior to or during the automatic retry.

7.5.1 Bus Errors

The bus error signal can be used to abort the bus cycle and the instruction being executed. BERR takes precedence over DSACKx provided it meets the timing constraints described in **SECTION 12 ELECTRICAL CHARACTERISTICS**. If BERR does not meet these constraints, it may cause unpredictable operation of the MC68020. If BERR remains asserted into the next bus cycle, it may cause incorrect operation of that cycle.

When the bus error signal is issued to terminate a bus cycle, the MC68020 may enter exception processing immediately following the bus cycle, or it may defer processing the exception. The instruction prefetch mechanism requests instruction words from the bus controller and the instruction cache before it is ready to execute them. If a bus error occurs on an instruction fetch, the processor does not take the exception until it attempts to use that instruction word. Should an intervening instruction cause a branch, or should a task switch occur, the bus error exception does not occur.

The bus error signal is recognized during a bus cycle in any of the following cases:

- DSACKx and HALT are negated and BERR is asserted.
- HALT and BERR are negated and DSACKx is asserted. BERR is then asserted within one clock cycle (HALT remains negated).
- BERR and HALT are asserted.

When the processor recognizes a bus error condition, it terminates the current bus cycle in the normal way. Figure 7-38 shows the timing of a bus error for the case in which DSACKx is not asserted. Figure 7-39 shows the timing for a bus error that is asserted after DSACKx. Exceptions are taken in both cases. (Refer to **8.1.2 Bus Error Exception** for details of bus error exception processing.) When BERR is asserted during a read cycle that supplies an instruction to the on-chip cache, the instruction in the cache is marked invalid.

In the second case, where BERR is asserted after DSACKx is asserted, BERR must be asserted within specification #48 (refer to **SECTION 12 ELECTRICAL CHARACTERISTICS**) for purely asynchronous operation, or it must be asserted and remain stable during the sample window, defined by specifications #27A and #47B, around the next falling edge of the clock after DSACKx is recognized. If BERR is not stable at this time, the processor may exhibit erratic behavior. BERR has priority over DSACKx. In this case, data may be present on the bus, but may not be valid. This sequence may be used by systems that have memory error detection and correction logic and by external cache memories.

7.5.2 Retry Operation

When the BERR and HALT signals are both asserted by an external device during a bus cycle, the processor enters the retry sequence. A delayed retry can also occur, similar to the delayed bus error signal described previously.

The processor terminates the bus cycle, places the control signals in their inactive state and does not begin another bus cycle until the BERR and HALT signals are negated by external logic. After a synchronization delay, the processor retries the previous cycle using the same access information (address, function code, size, etc.) The BERR signal should be negated before S2 of the read cycle in order to ensure correct operation of the retried cycle. Figure 7-40 shows a retry operation of a cycle.

The processor retries any read or write cycle of a read-modify-write operation separately; the read-modify-write cycle signal (RMC) remains asserted during the entire retry sequence.

Asserting the bus request signal (BR) along with BERR and HALT provides a relinquish and retry operation. The MC68020 does not relinquish the bus during a read-modify-write operation. Any device that requires the processor to give up the bus and retry a bus cycle

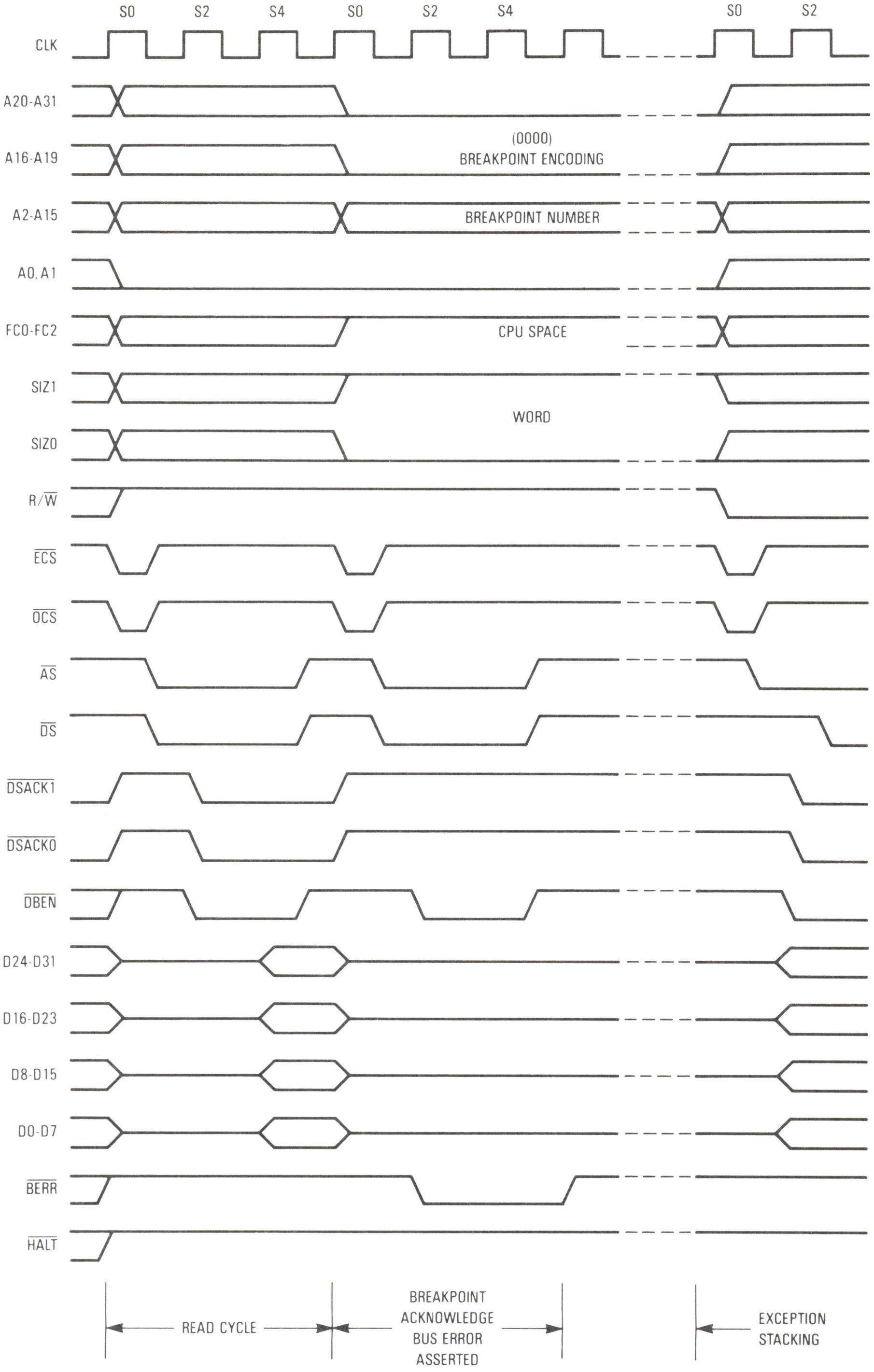

S0
S2
S4
S0
S2
S4
S0
S2
CLK
A20-A31
A16-A19
(0000)
BREAKPOINT ENCODING
A2-A15
BREAKPOINT NUMBER
A0, A1
FC0-FC2
CPU SPACE
SIZ1
WORD
SIZ0
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D24-D31
D16-D23
D8-D15
D0-D7
$\overline{BERR}$
$\overline{HALT}$
READ CYCLE
BREAKPOINT
ACKNOWLEDGE
BUS ERROR
ASSERTED
EXCEPTION
STACKING

Figure 7-38. Bus Error without $\overline{\text{DSACKx}}$

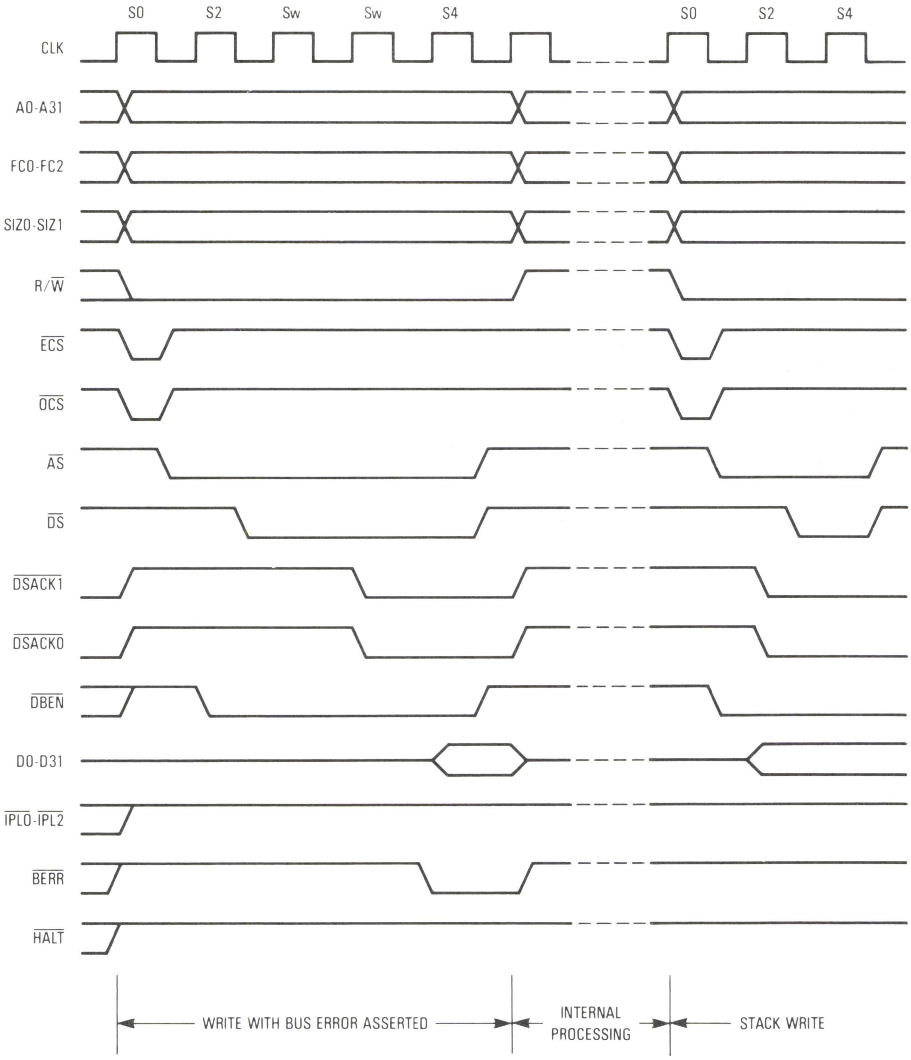

S0
S2
Sw
Sw
S4
S0
S2
S4
CLK
A0-A31
FC0-FC2
SIZ0-SIZ1
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D0-D31
$\overline{IPL0}$-$\overline{IPL2}$
$\overline{BERR}$
$\overline{HALT}$
WRITE WITH BUS ERROR ASSERTED
INTERNAL PROCESSING
STACK WRITE

Figure 7-39. Late Bus Error with $\overline{\text{DSACKx}}$

during a read-modify-write cycle must assert $\overline{\text{BERR}}$ and $\overline{\text{BR}}$ only ($\overline{\text{HALT}}$ must not be included). The bus error handler software should examine the read-modify-write bit in the special status word (refer to **8.2.1 Special Status Word**) and take the appropriate action to resolve this type of fault when it occurs.

7.5.3 Halt Operation

When the halt signal ($\overline{\text{HALT}}$) is asserted and $\overline{\text{BERR}}$ is not asserted, the MC68020 halts external bus activity at the next bus cycle boundary. $\overline{\text{HALT}}$ by itself does not terminate a

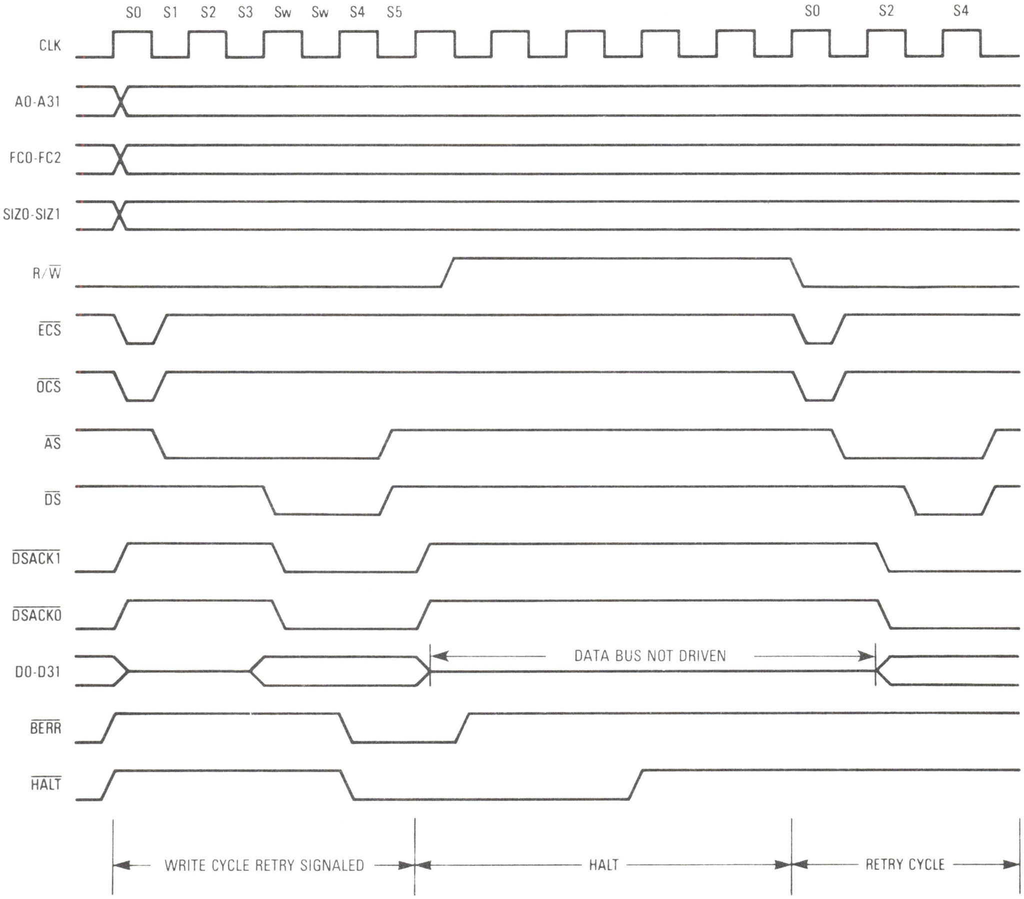

S0
S1
S2
S3
Sw
Sw
S4
S5
S0
S2
S4
CLK
A0-A31
FC0-FC2
SIZ0-SIZ1
R/W̄
ECS̄
OCS̄
AS̄
DS̄
DSACK1̄
DSACK0̄
D0-D31
DATA BUS NOT DRIVEN
BERR̄
HALT̄
WRITE CYCLE RETRY SIGNALED
HALT
RETRY CYCLE

Figure 7-40. Late Retry

bus cycle. Negating and reasserting $\overline{\text{HALT}}$ in accordance with the correct timing requirements provides a single step (bus cycle to bus cycle) operation. The $\overline{\text{HALT}}$ signal affects external bus cycles only, so a program that resides in the instruction cache and does not require use of the external bus may continue executing, unaffected by the $\overline{\text{HALT}}$ signal.

The single-cycle mode allows the user to proceed through (and debug) external processor operations, one bus cycle at a time. Figure 7-41 shows the timing requirements for a single-cycle operation. Since the occurrence of a bus error while $\overline{\text{HALT}}$ is asserted causes a retry operation, the user must anticipate retry cycles while debugging in the single-cycle mode. The single-step operation and the software trace capability allow the system debugger to trace single bus cycles, single instructions, or changes in program flow. These processor capabilities, along with a software debugging package, give complete debugging flexibility.

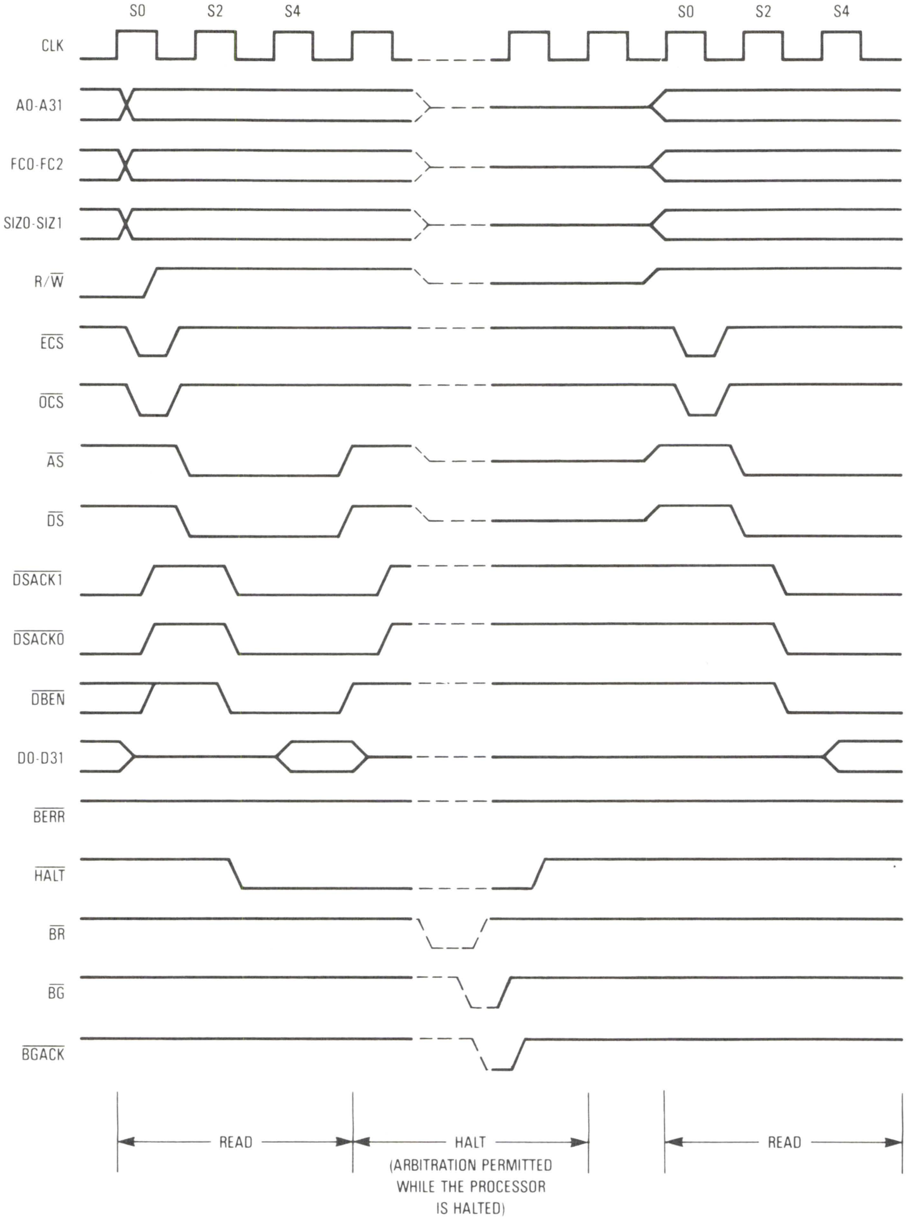

S0
S2
S4
S0
S2
S4
CLK
A0-A31
FC0-FC2
SIZ0-SIZ1
R/W̄
ECS̄
OCS̄
AS̄
DS̄
DSACK1̄
DSACK0̄
DBEN̄
D0-D31
BERR̄
HALT̄
BR̄
BḠ
BGACK̄
READ
HALT
(ARBITRATION PERMITTED WHILE THE PROCESSOR IS HALTED)
READ

Figure 7-41. Halt Operation Timing

When the processor completes a bus cycle with the $\overline{\text{HALT}}$ signal asserted, the data bus (D31–D0) is placed in the high-impedance state, and bus control signals are driven inactive (not high-impedance state); the address, function code, size, and read/write signals remain in the same state. The halt operation has no effect on bus arbitration (refer to **7.7 BUS ARBITRATION**). When bus arbitration occurs while the MC68020 is halted, the address and control signals are also placed in the high-impedance state. Once bus mastership is returned to the MC68020, if HALT is still asserted, the address, function code, size, and read/write signals are again driven to their previous states. The processor does not service interrupt requests while it is halted, but it may assert the IPEND signal as appropriate.

7.5.4 Double Bus Fault

When a bus error or an address error occurs during the exception processing sequence for a previous bus error, a previous address error, or a reset exception, the bus or address error causes a double bus fault. For example, the processor attempts to stack several words containing information about the state of the machine while processing a bus error exception. If a bus error exception occurs during the stacking operation, the second error is considered a double bus fault. Only an external reset operation can restart a halted processor. However, bus arbitration can still occur (refer to **7.7 BUS ARBITRATION**). When a double bus fault occurs, the processor halts and drives the HALT line low.

A second bus error or address error that occurs after exception processing has completed (during the execution of the exception handler routine, or later) does not cause a double bus fault. A bus cycle that is retried does not constitute a bus error or contribute to a double bus fault either. The processor continues to retry the same bus cycle as long as the external hardware requests it.

7.6 BUS SYNCHRONIZATION

The MC68020 overlaps instruction execution; that is, during bus activity for one instruction, instructions that do not use the external bus can be executed. Due to the independent operation of the on-chip cache relative to the operation of the bus controller, many subsequent instructions can be executed, resulting in seemingly nonsequential instruction execution. When this is not desired and the system depends on sequential execution following bus activity, the NOP instruction can be used. The NOP instruction forces instruction and bus synchronization in that it freezes instruction execution until all pending bus cycles have completed.

An example of the use of the NOP instruction for this purpose is the case of a write operation of control information to an external register, where the external hardware attempts to control program execution based on the data that is written with the conditional assertion of BERR. Since the MC68020 cannot process the bus error until the end of the bus cycle, the external hardware has not successfully interrupted program execution. In order to prevent a subsequent instruction from executing until the external cycle completes, a NOP instruction can be inserted after the instruction causing the write. In this case, bus error exception processing proceeds immediately after the write before subsequent instructions are executed. This is an irregular situation, and the use of the NOP instruction for this purpose is not required by most systems.

7.7 BUS ARBITRATION

The bus design of the MC68020 provides for a single bus master at any one time: either the processor or an external device. One or more of the external devices on the bus can have the capability of becoming bus master. Bus arbitration is the protocol by which an external device becomes bus master; the bus controller in the MC68020 manages the bus arbitration signals so that the processor has the lowest priority. External devices that need to obtain the bus must assert the bus arbitration signals in the sequences described in the following paragraphs. Systems that include several devices that can become bus master require external circuitry to assign priorities to the devices, so that when two or more external devices attempt to become bus master at the same time, the one having the highest priority becomes bus master first. The sequence of the protocol is:

1. An external device asserts the bus request signal.
2. The processor asserts the bus grant signal to indicate that the bus becomes available at the end of the current bus cycle.
3. The external device asserts the bus grant acknowledge signal to indicate that it has assumed bus mastership.

The bus request signal ($\overline{\text{BR}}$) may be issued any time during a bus cycle, or between cycles. The bus grant signal ($\overline{\text{BG}}$) is asserted in response to $\overline{\text{BR}}$; it is usually asserted as soon as $\overline{\text{BR}}$ has been synchronized and recognized, except when the MC68020 has made an internal decision to execute a bus cycle. Then, the assertion of $\overline{\text{BG}}$ is deferred until the bus cycle has begun. Additionally, $\overline{\text{BG}}$ is not asserted until the end of a read-modify-write operation (when $\overline{\text{RMC}}$ is negated) in response to a $\overline{\text{BR}}$ signal. When the requesting device receives $\overline{\text{BG}}$ and more than one external device can be bus master, the requesting device should begin whatever arbitration is required. The external device asserts the bus grant acknowledge signal ($\overline{\text{BGACK}}$) when it assumes bus mastership, and maintains $\overline{\text{BGACK}}$ during the entire bus cycle (or cycles) for which it is bus master. The following conditions must be met for an external device to assume mastership of the bus through the normal bus arbitration procedure:

- It must have received $\overline{\text{BG}}$ through the arbitration process.
- Address strobe ($\overline{\text{AS}}$) must be negated indicating that no bus cycle is in progress, and the external device must ensure that all appropriate processor signals have been placed in the high-impedance state (by observing specification #7 in **SECTION 12 ELECTRICAL CHARACTERISTICS**).
- The termination signal ($\overline{\text{DSACKx}}$) for the most recent cycle must have become inactive, indicating that external devices are off the bus (optional, refer to **7.7.3 Bus Grant Acknowledge**).
- $\overline{\text{BGACK}}$ must be inactive, indicating that no other bus master has claimed ownership of the bus.

Figure 7-42 is a flowchart showing the detail involved in bus arbitration for a single device. Figure 7-43 is a timing diagram for the same operation. This technique allows processing of bus requests during data transfer cycles.

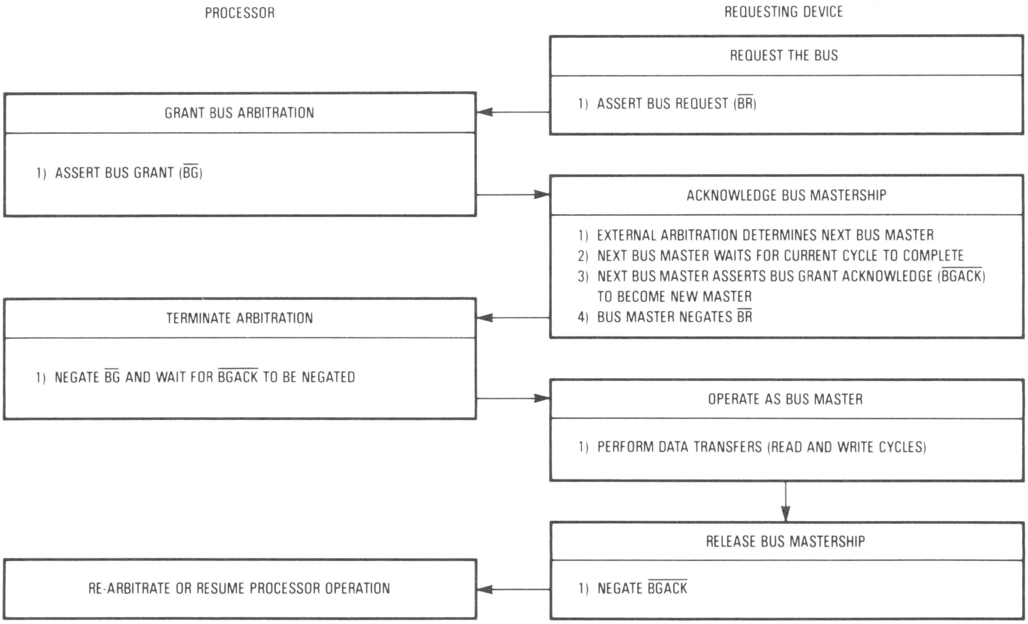

PROCESSOR
REQUESTING DEVICE
REQUEST THE BUS
1) ASSERT BUS REQUEST ($\overline{BR}$)
GRANT BUS ARBITRATION
1) ASSERT BUS GRANT ($\overline{BG}$)
ACKNOWLEDGE BUS MASTERSHIP
1) EXTERNAL ARBITRATION DETERMINES NEXT BUS MASTER
2) NEXT BUS MASTER WAITS FOR CURRENT CYCLE TO COMPLETE
3) NEXT BUS MASTER ASSERTS BUS GRANT ACKNOWLEDGE ($\overline{BGACK}$) TO BECOME NEW MASTER
4) BUS MASTER NEGATES $\overline{BR}$
TERMINATE ARBITRATION
1) NEGATE $\overline{BG}$ AND WAIT FOR $\overline{BGACK}$ TO BE NEGATED
OPERATE AS BUS MASTER
1) PERFORM DATA TRANSFERS (READ AND WRITE CYCLES)
RELEASE BUS MASTERSHIP
1) NEGATE $\overline{BGACK}$
RE-ARBITRATE OR RESUME PROCESSOR OPERATION

Figure 7-42. Bus Arbitration Flowchart for Single Request

The timing diagram shows that $\overline{BR}$ is negated at the time that $\overline{BGACK}$ is asserted. This type of operation applies to a system consisting of the processor and one device capable of bus mastership. In a system having a number of devices capable of bus mastership, the bus request line from each device can be wire-ORed to the processor. In such a system, more than one bus request could be asserted simultaneously.

The timing diagram in Figure 7-43 shows that $\overline{BG}$ is negated a few clock cycles after the transition of the bus grant acknowledge signal. However, if bus requests are still pending after the negation of bus grant, the processor asserts another bus grant within a few clock cycles after it was negated. This additional assertion of bus grant allows external arbitration circuitry to select the next bus master before the current bus master has finished with the bus. The following paragraphs provide additional information about the three steps in the arbitration process.

Bus arbitration requests are recognized during normal processing, $\overline{RESET}$ assertion, $\overline{HALT}$ assertion, and even when the processor has halted due to a double bus fault.

7.7.1 Bus Request

External devices capable of becoming bus masters request the bus by asserting the bus request ($\overline{BR}$) signal. This can be a wire-ORed signal (although it need not be constructed from open-collector devices) that indicates to the processor that some external device requires control of the bus. The processor is effectively at a lower bus priority level than

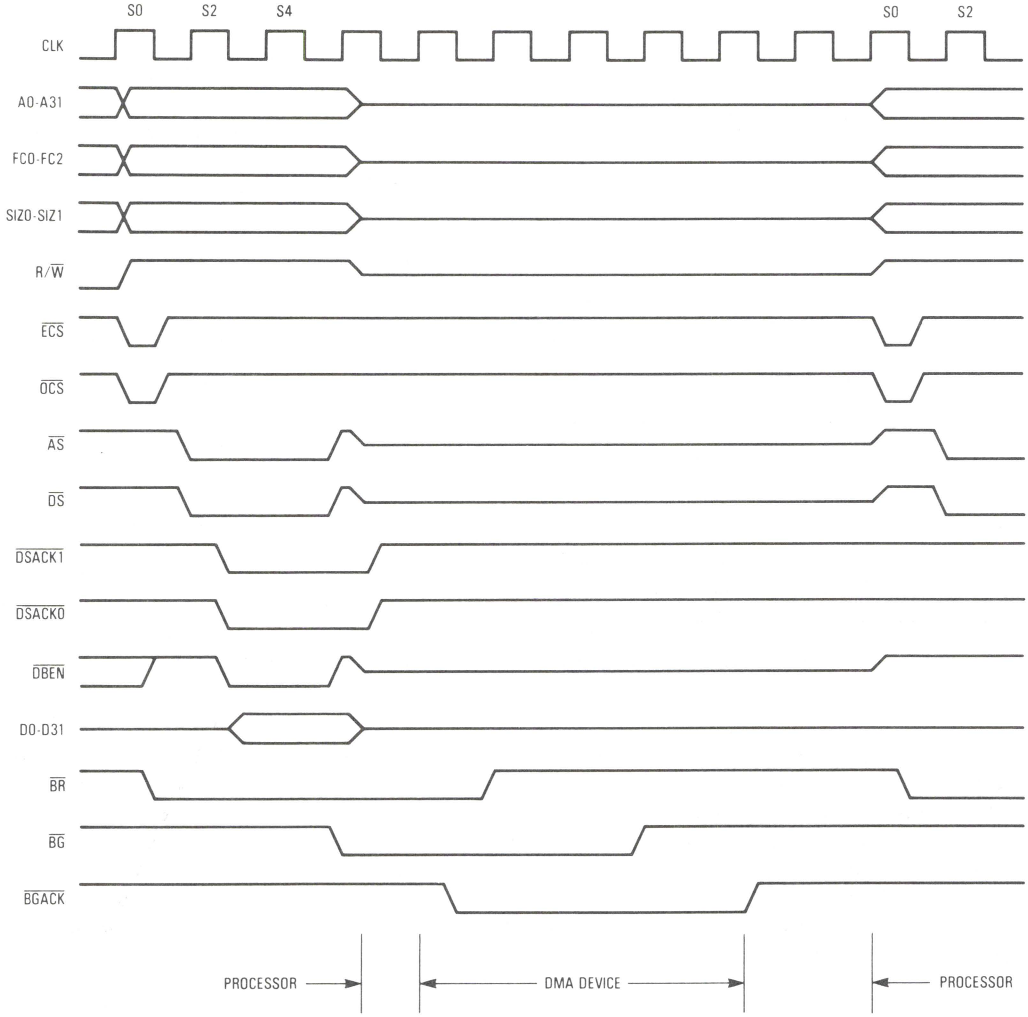

S0
S2
S4
S0
S2
CLK
A0-A31
FC0-FC2
SIZ0-SIZ1
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D0-D31
$\overline{BR}$
$\overline{BG}$
$\overline{BGACK}$
PROCESSOR
DMA DEVICE
PROCESSOR

Figure 7-43. Bus Arbitration Operation Timing

the external device and relinquishes the bus after it has completed the current bus cycle (if one has started).

If no acknowledge is received while the bus request signal is active, the processor remains bus master once the bus request is negated. This prevents unnecessary interference with ordinary processing if the arbitration circuitry inadvertently responds to noise or an external device determines that it no longer requires use of the bus before it has been granted mastership.

7.7.2 Bus Grant

The processor asserts bus grant ($\overline{\mathrm{BG}}$) as soon as possible after receipt of the bus request. This is immediately following internal synchronization except during a read-modify-write cycle or following an internal decision to execute a bus cycle. During a read-modify-write cycle, the processor does not assert bus grant until the entire operation has completed. The read-modify-write cycle signal ($\overline{\mathrm{RMC}}$) is asserted to indicate that the bus is locked. In the case of an internal decision to execute another bus cycle, $\overline{\mathrm{BG}}$ is deferred until the bus cycle has begun.

The bus grant signal may be routed through a daisy-chained network or through a specific priority-encoded network. The processor allows any type of external arbitration that follows the protocol.

7.7.3 Bus Grant Acknowledge

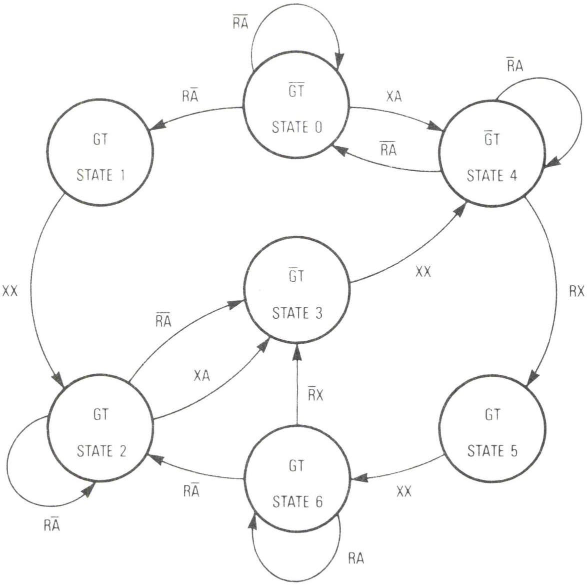
Upon receiving $\overline{\mathrm{BG}}$, the requesting device waits until address strobe ($\overline{\mathrm{AS}}$), data transfer and size acknowledge ($\overline{\mathrm{DSACKx}}$), and bus grant acknowledge ($\overline{\mathrm{BGACK}}$) are negated before asserting its own $\overline{\mathrm{BGACK}}$. The negation of the $\overline{\mathrm{AS}}$ indicates that the previous master releases the bus after specification #7 (refer to **SECTION 12 ELECTRICAL CHARACTERISTICS**). The negation of $\overline{\mathrm{DSACKx}}$ indicates that the previous slave has completed its cycle with the previous master. Note that in some applications $\overline{\mathrm{DSACKx}}$ might not be used in this way.

General-purpose devices are then connected to be dependent only on $\overline{\mathrm{AS}}$. When $\overline{\mathrm{BGACK}}$ is asserted, the device is the bus master until it negates $\overline{\mathrm{BGACK}}$. Bus grant acknowledge should not be negated until all bus cycles required by the alternate bus master are completed. Bus mastership terminates at the negation of $\overline{\mathrm{BGACK}}$. The bus request from the granted device should be negated after $\overline{\mathrm{BGACK}}$ is asserted. If a bus request is still pending after the assertion of $\overline{\mathrm{BGACK}}$, another bus grant is asserted within a few clocks of the negation of the bus grant, as described in the **7.7.4 Bus Arbitration Control**. Note that the processor does not perform any external bus cycles before it reasserts bus grant in this case.

7.7.4 Bus Arbitration Control

The bus arbitration control unit in the MC68020 is implemented with a finite state machine. As discussed previously, all asynchronous inputs to the MC68020 are internally synchronized in a maximum of two cycles of the processor clock.

As shown in Figure 7-44, input signals labeled R and A are internally synchronized versions of the bus request and bus grant acknowledge signals respectively. The bus grant output is labeled G and the internal high-impedance control signal is labeled T. If T is true, the address, data, and control buses are placed in the high-impedance state after the next rising edge following the negation of address strobe ($\overline{\mathrm{AS}}$) and the read-modify-write-cycle ($\overline{\mathrm{RMC}}$) signal. All signals are shown in positive logic (active high) regardless of their true active voltage level.


$\overline{R}\overline{A}$
$\overline{G}\overline{T}$
STATE 0
$R\overline{A}$
GT
STATE 1
XA
$\overline{R}\overline{A}$
$\overline{R}A$
$\overline{G}T$
STATE 4
XX
$\overline{G}T$
STATE 3
XX
RX
$\overline{R}\overline{A}$
XA
$\overline{R}X$
GT
STATE 2
GT
STATE 5
$R\overline{A}$
GT
STATE 6
XX
$R\overline{A}$
RA

R - BUS REQUEST
A - BUS GRANT ACKNOWLEDGE
G - BUS GRANT
T - THREE-STATE CONTROL TO BUS CONTROL LOGIC
X - DON'T CARE

NOTE: The $\overline{BG}$ output will not be asserted while $\overline{RMC}$ is asserted.

Figure 7-44. Bus Arbitration State Diagram

State changes occur on the next rising edge of the clock after the internal signal is valid. The $\overline{BG}$ signal transitions on the falling edge of the clock after a state is reached during which G changes. The bus control signals (controlled by T) are driven by the processor immediately following a state change, when bus mastership is returned to the MC68020.

State 0, at the top center of the diagram, in which G and T are both negated, is the state of the bus arbiter while the processor is bus master. Request R and acknowledge A keep the arbiter in state 0 as long as they are both negated. When a request R is received, both grant G and signal T are asserted (in state 1 at the top left). The next clock causes a change to state 2 at the lower left, in which G and T are held. The bus arbiter remains in that state until acknowledge A is asserted or request R is negated. Once either occurs, the arbiter changes to the center state, state 3, and negates grant G. The next clock takes the arbiter to state 4, at the upper right, in which grant G remains negated and signal T remains asserted. With acknowledge A asserted, the arbiter remains in state 4 until A is negated or request R is again asserted. When A is negated, the arbiter returns to the original state, state 0, and negates signal T. This sequence of states follows the normal sequence of signals for relinquishing the bus to an external bus master. Other states apply to other possible sequences of combinations of R and A.

The MC68020 does not allow arbitration of the external bus during the RMW sequence. For the duration of this sequence the MC68020 ignores the $\overline{BR}$ input. If mastership of the MC68020 bus is required during an RMW operation, the bus error signal must be used to abort the RMW sequence. The bus arbitration sequence while the bus is inactive (i.e., executing internal operations such as a multiply instruction) is shown in Figure 7-45.

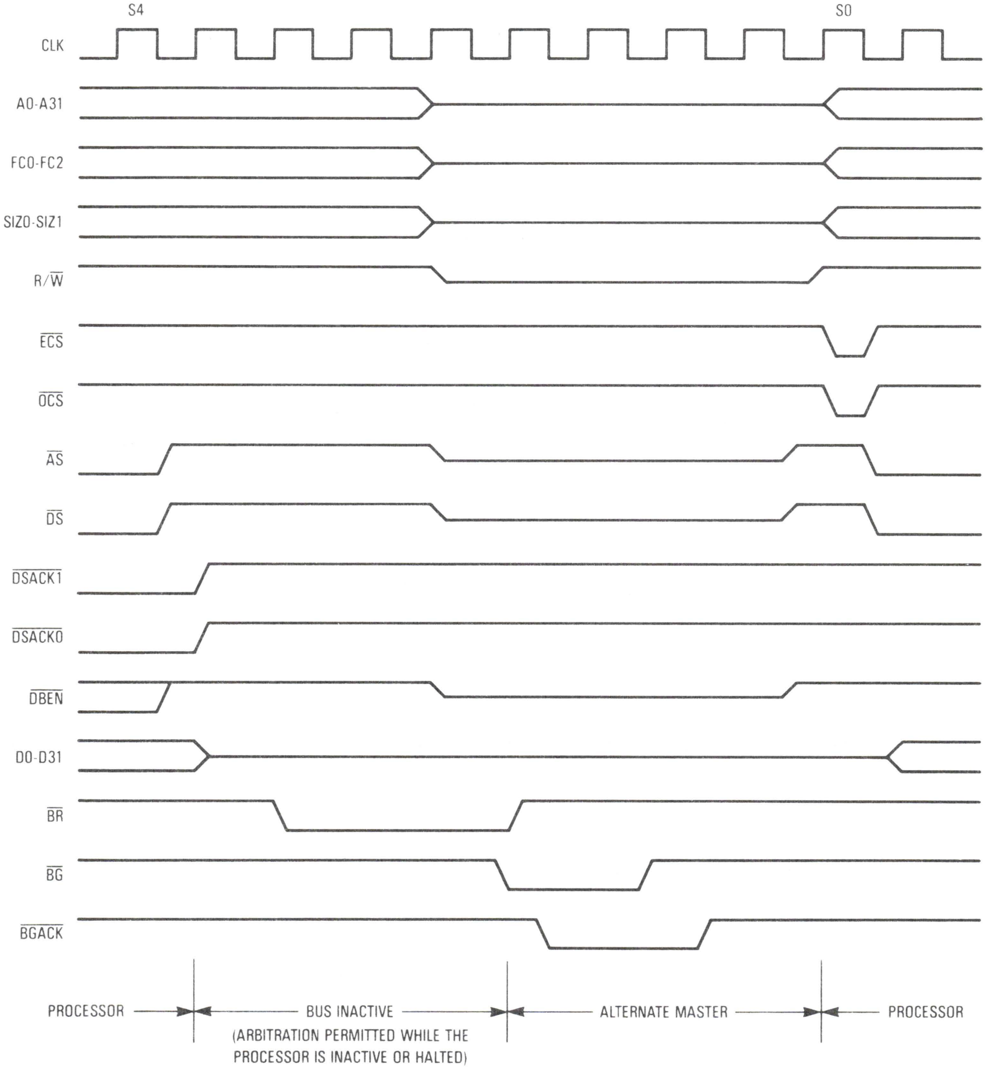

S4
S0
CLK
A0-A31
FC0-FC2
SIZ0-SIZ1
R/W
ECS
OCS
AS
DS
DSACK1
DSACK0
DBEN
D0-D31
BR
BG
BGACK
PROCESSOR
BUS INACTIVE
(ARBITRATION PERMITTED WHILE THE PROCESSOR IS INACTIVE OR HALTED)
ALTERNATE MASTER
PROCESSOR

Figure 7-45. Bus Arbitration Operation (Bus Inactive)

7.8 RESET OPERATION

The reset signal ($\overline{\text{RESET}}$) is a bidirectional signal with which an external device resets the system or the processor resets external devices. When power is applied to the system, external circuitry should assert $\overline{\text{RESET}}$ for a minimum of 100 ms. after V_{CC} is within tolerance. Figure 7-46 is a timing diagram of the power-up reset operation, showing the relationships between $\overline{\text{RESET}}$, V_{CC}, and bus signals. The clock signal is required to be stable by the time V_{CC} reaches the minimum operating specification. During the reset period, the entire bus (except for non-three-statable signals, which are driven to their inactive state) three-states. Once $\overline{\text{RESET}}$ negates, all control signals are driven to their inactive state, the data bus is in read mode, and the address bus is driven. After this, the first bus cycle for RESET exception processing begins.

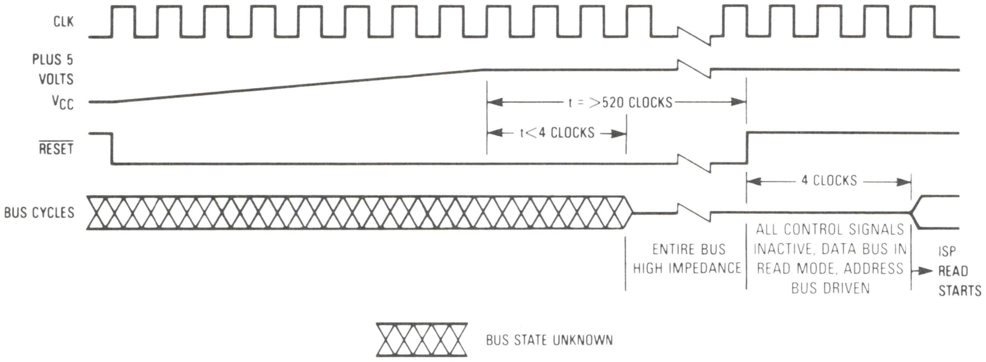

CLK
PLUS 5 VOLTS
V_{CC}
$\overline{\text{RESET}}$
BUS CYCLES
t = >520 CLOCKS
t<4 CLOCKS
4 CLOCKS
ENTIRE BUS HIGH IMPEDANCE
ALL CONTROL SIGNALS INACTIVE, DATA BUS IN READ MODE, ADDRESS BUS DRIVEN
ISP READ STARTS
BUS STATE UNKNOWN

Figure 7-46. Initial Reset Operation Timing

The external $\overline{\text{RESET}}$ signal resets the processor and the entire system. Except for the initial reset, $\overline{\text{RESET}}$ should be asserted for at least 520 clock periods to ensure that the processor resets. Asserting $\overline{\text{RESET}}$ for ten clock periods is sufficient for resetting the processor logic; the additional clock periods prevent a reset instruction from overlapping the external $\overline{\text{RESET}}$ signal.

Resetting the processor causes any bus cycle in progress to terminate as if $\overline{\text{DSACKx}}$ or $\overline{\text{BERR}}$ had been asserted. In addition, the processor initializes registers appropriately for a reset exception. Exception processing for a reset operation is described in **8.1.1 Reset Exception**.

When a reset instruction is executed, the processor drives the $\overline{\text{RESET}}$ signal for 512 clock cycles. In this case, the processor resets the external devices of the system, and the internal registers of the processor are unaffected. The external devices connected to the $\overline{\text{RESET}}$ signal are reset at the completion of the reset instruction. An external $\overline{\text{RESET}}$ signal that is asserted to the processor during execution of a reset instruction must extend beyond the reset period of the instruction by at least eight clock cycles in order to reset the processor. Figure 7-47 shows the timing information for the reset instruction.

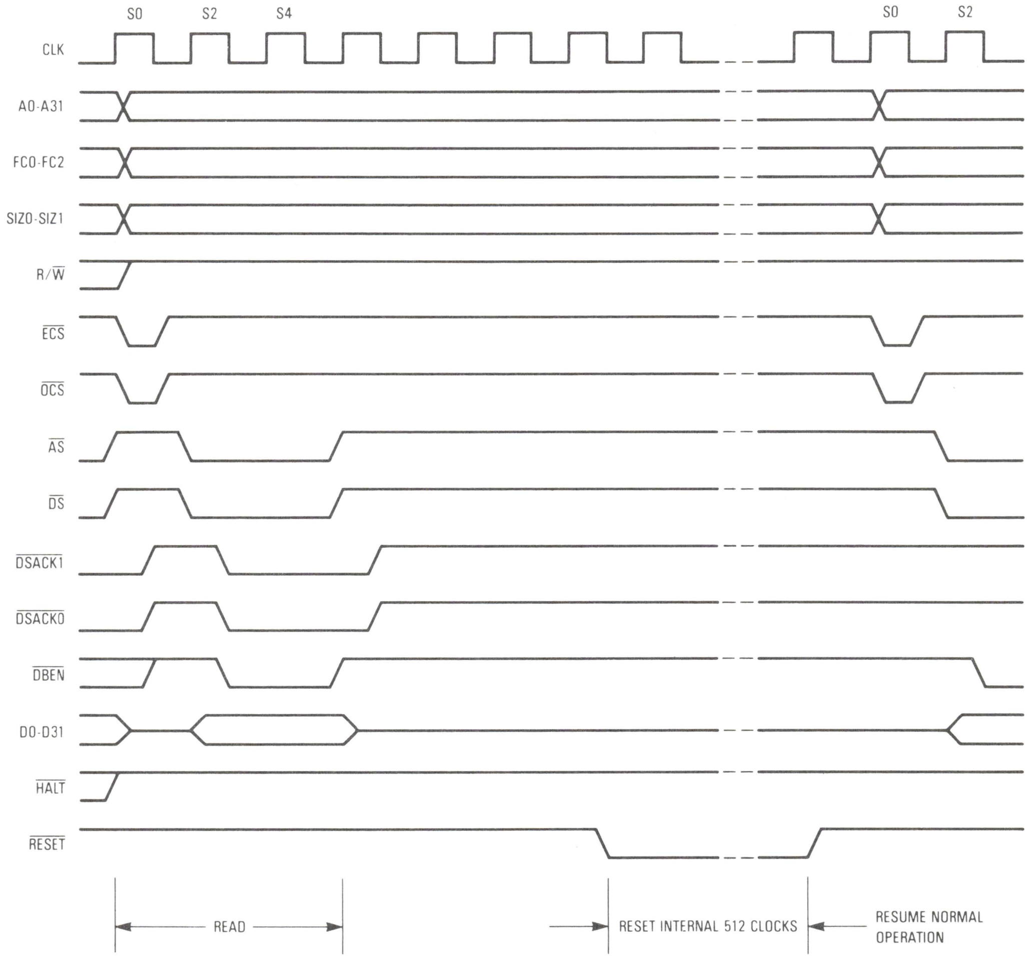

S0
S2
S4
S0
S2
CLK
A0-A31
FC0-FC2
SIZ0-SIZ1
R/$\overline{W}$
$\overline{ECS}$
$\overline{OCS}$
$\overline{AS}$
$\overline{DS}$
$\overline{DSACK1}$
$\overline{DSACK0}$
$\overline{DBEN}$
D0-D31
$\overline{HALT}$
$\overline{RESET}$
READ
RESET INTERNAL 512 CLOCKS
RESUME NORMAL OPERATION

Figure 7-47. Processor Generated Reset Operation

SECTION 8
EXCEPTION PROCESSING

Exception processing is defined as the activities performed by the processor in preparing to execute a handler routine for any condition that causes an exception. In particular, exception processing does not include execution of the handler routine itself. An introduction to exception processing, as one of the processing states of the MC68020 processor, was given in **SECTION 4 PROCESSING LEVELS**. This section describes exception processing in detail, describing the processing for each type of exception. It describes the return from an exception and bus fault recovery. This section also describes the formats of the exception stack frames. For more detail on protocol violation and coprocessor-related exceptions, refer to **SECTION 9 COPROCESSOR INTERFACE DESCRIPTION**. Also, for more detail on exceptions defined for floating-point coprocessors, refer to the user's manual for the MC68881/MC68882.

8.1 EXCEPTION PROCESSING SEQUENCE

Exception processing occurs in four functional steps. However, all individual bus cycles associated with exception processing (vector acquisition, stacking, etc.) are not guaranteed to occur in the order in which they are described in this section. Nonetheless, all addresses and offsets from the stack pointer are guaranteed to be as described.

The first step of exception processing involves the status register. The processor makes an internal copy of the status register. Then the processor sets the S bit, changing to the supervisor privilege level. Next, the processor inhibits tracing of the exception handler by clearing the T1 and T0 bits. For the reset and interrupt exceptions, the processor also updates the interrupt priority mask.

In the second step, the processor determines the vector number of the exception. For interrupts, the processor performs an interrupt acknowledge cycle (a read from the CPU address space type $F; see Figures 7-34 and 7-35) to obtain the vector number. For coprocessor-detected exceptions, the vector number is included in the coprocessor exception primitive response. (Refer to **SECTION 9 COPROCESSOR INTERFACE DESCRIPTION** for a complete discussion of coprocessor exceptions.) For all other exceptions, internal logic provides the vector number. This vector number is used in the last step to calculate the address of the exception vector. Throughout this section, vector numbers are given in decimal notation.

For all exceptions other than reset, the third step is to save the current processor context. The processor creates an exception stack frame on the active supervisor stack and fills it with context information appropriate for the type of exception. Other information may also be stacked, depending on which exception is being processed and the state of the processor prior to the exception. If the exception is an interrupt and the M bit of the status register is set, the processor clears the M bit in the status register and builds a second stack frame on the interrupt stack.

The last step initiates execution of the exception handler. The processor multiplies the vector number by four to determine the exception vector offset. It adds the offset to the value stored in the vector base register to obtain the memory address of the exception vector. Next, the processor loads the program counter (and the interrupt stack pointer (ISP) for the reset exception) from the exception vector table in memory. After prefetching the first three words to fill the instruction pipe, the processor resumes normal processing at the address in the program counter. Table 8-1 contains a description of all the exception vector offsets defined for the MC68020.

As shown in Table 8-1, the first 64 vectors are defined by Motorola and 192 vectors are reserved for interrupt vectors defined by the user. However, external devices may use vectors reserved for internal purposes at the discretion of the system designer.

8.1.1 Reset Exception

Assertion by external hardware of the $\overline{\text{RESET}}$ signal causes a reset exception. For details on the requirements for the assertion of $\overline{\text{RESET}}$, refer to **7.8 RESET OPERATION**.

The reset exception has the highest priority of any exception; it provides for system initialization and recovery from catastrophic failure. It aborts any processing in progress when it is recognized, and that processing cannot be recovered. Figure 8-1 is a flowchart of the reset exception, which performs the following operations:

1. Clears both trace bits in the status register to disable tracing.
2. Places the processor in the interrupt mode of the supervisor privilege level by setting the supervisor bit and clearing the master bit in the status register.
3. Sets the processor interrupt priority mask to the highest priority level (level seven).
4. Initializes the vector base register to zero ($00000000).
5. Clears the enable and freeze bits for the on-chip cache.
6. Invalidates all entries in the instruction cache.
7. Generates a vector number to reference the reset exception vector (two long words) at offset zero in the supervisor program address space.
8. Loads the first long word of the reset exception vector into the interrupt stack pointer.
9. Loads the second long word of the reset exception vector into the program counter.

After the initial instruction prefetches, program execution begins at the address in the program counter. The reset exception does not save the value of either the program counter or the status register.

As described in **7.5.4 Double Bus Fault**, if bus error or address error occur during the exception processing sequence for a reset, a double bus fault occurs. The processor halts and asserts the $\overline{\text{HALT}}$ signal to indicate the halted condition.

Execution of the reset instruction does not cause a reset exception, nor does it affect any internal registers, but it does cause the MC68020 to assert the $\overline{\text{RESET}}$ signal, resetting all external devices.

Table 8-1. Exception Vector Assignments

Vector Number(s)	Vector Offset		Assignment
	Hex	Space	
0	000	SP	Reset Initial Interrupt Stack Pointer
1	004	SP	Reset Initial Program Counter
2	008	SD	Bus Error
3	00C	SD	Address Error
4	010	SD	Illegal Instruction
5	014	SD	Zero Divide
6	018	SD	CHK, CHK2 Instruction
7	01C	SD	cpTRAPcc, TRAPcc, TRAPV Instructions
8	020	SD	Privilege Violation
9	024	SD	Trace
10	028	SD	Line 1010 Emulator
11	02C	SD	Line 1111 Emulator
12	030	SD	(Unassigned, Reserved)
13	034	SD	Coprocessor Protocol Violation
14	038	SD	Format Error
15	03C	SD	Uninitialized Interrupt
16 Through 23	040 05C	SD SD	Unassigned, Reserved
24	060	SD	Spurious Interrupt
25	064	SD	Level 1 Interrupt Autovector
26	068	SD	Level 2 Interrupt Autovector
27	06C	SD	Level 3 Interrupt Autovector
28	070	SD	Level 4 Interrupt Autovector
29	074	SD	Level 5 Interrupt Autovector
30	078	SD	Level 6 Interrupt Autovector
31	07C	SD	Level 7 Interrupt Autovector
32 Through 47	080 0BC	SD SD	TRAP #0-15 Instruction Vectors
48	0C0	SD	FPCP Branch or Set on Unordered Condition
49	0C4	SD	FPCP Inexact Result
50	0C8	SD	FPCP Divide by Zero
51	0CC	SD	FPCP Underflow
52	0D0	SD	FPCP Operand Error
53	0D4	SD	FPCP Overflow
54	0D8	SD	FPCP Signaling NAN
55	0DC	SD	Unassigned, Reserved
56	0E0	SD	PMMU Configuration
57	0E4	SD	PMMU Illegal Operation
58	0E8	SD	PMMU Access Level Violation
59 Through 63	0EC 0FC	SD SD	Unassigned, Reserved
64 Through 255	100 3FC	SD SD	User-Defined Vectors (192)

SP = Supervisor Program Space SD = Supervisor Data Space

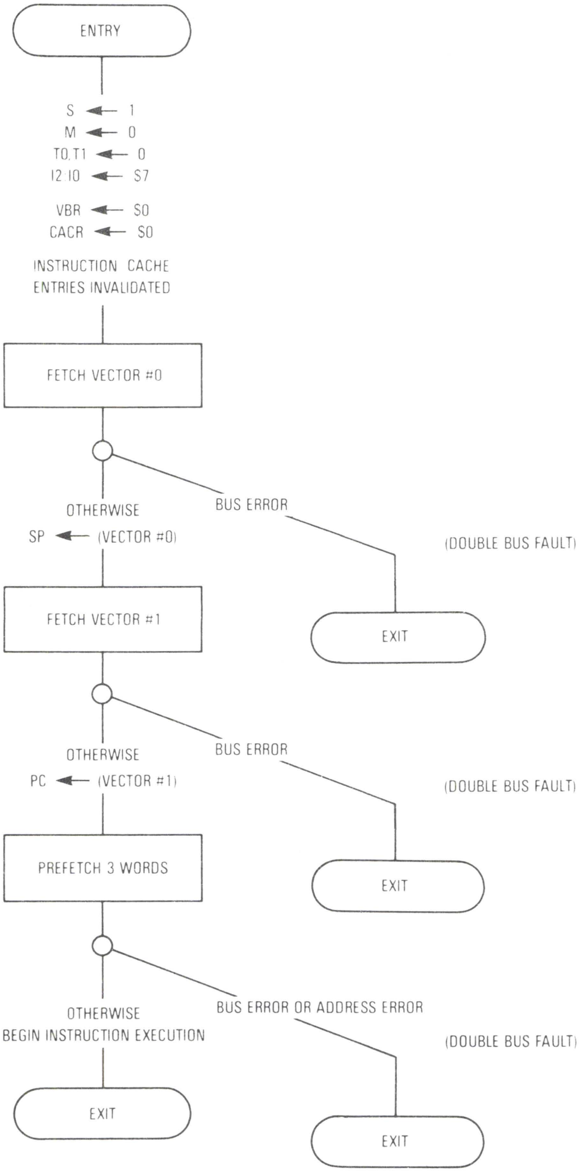

ENTRY
S ← 1
M ← 0
T0,T1 ← 0
I2:I0 ← $7
VBR ← $0
CACR ← $0
INSTRUCTION CACHE ENTRIES INVALIDATED
FETCH VECTOR #0
BUS ERROR
OTHERWISE
SP ← (VECTOR #0)
(DOUBLE BUS FAULT)
EXIT
FETCH VECTOR #1
BUS ERROR
OTHERWISE
PC ← (VECTOR #1)
(DOUBLE BUS FAULT)
EXIT
PREFETCH 3 WORDS
BUS ERROR OR ADDRESS ERROR
OTHERWISE
BEGIN INSTRUCTION EXECUTION
(DOUBLE BUS FAULT)
EXIT
EXIT

Figure 8-1. Reset Operation Flowchart

8.1.2 Bus Error Exception

A bus error exception occurs when external logic aborts a bus cycle by asserting the $\overline{\text{BERR}}$ input signal. If the aborted bus cycle is a data access, the processor immediately begins exception processing. If the aborted bus cycle is an instruction prefetch, the processor may delay taking the exception until it attempts to use the prefetched information.

The processor begins exception processing for a bus error by making an internal copy of the current status register. The processor then enters the supervisor privilege level (by setting the S bit in the status register) and clears the trace bits. The processor generates exception vector number 2 for the bus error vector. It saves the vector offset, program counter, and the internal copy of the status register on the stack. The saved program counter value is the logical address of the instruction that was executing at the time the fault was detected. This is not necessarily the instruction that initiated the bus cycle, since the processor overlaps execution of instructions. The processor also saves the contents of some of its internal registers. The information saved on the stack is sufficient to identify the cause of the bus fault and recover from the error.

For efficiency, the MC68020 uses two different bus error stack frame formats. When the bus error exception is taken at an instruction boundary, less information is required to recover from the error, and the processor builds the short bus fault stack frame as shown in Table 8-6. When the exception is taken during the execution of an instruction, the processor must save its entire state for recovery and uses the long bus fault stack frame shown in Table 8-6. The format code in the stack frame distinguishes the two stack frame formats. Stack frame formats are described in detail in **8.4 EXCEPTION STACK FRAME FORMATS**.

If a bus error occurs during the exception processing for a bus error, address error, or reset, or while the processor is loading internal state information from the stack during the execution of an RTE instruction, a double bus fault occurs, and the processor enters the halted state. In this case, the processor does not attempt to alter the current state of memory. Only an external $\overline{\text{RESET}}$ can restart a processor halted by a double bus fault.

8.1.3 Address Error Exception

An address error exception occurs when the processor attempts to prefetch an instruction from an odd address. This exception is similar to a bus error exception but is internally initiated. A bus cycle is not executed, and the processor begins exception processing immediately. After exception processing commences, the sequence is the same as that for bus error exceptions described in the preceding paragraphs, except that the vector number is 3 and the vector offset in the stack frame refers to the address error vector. Either a short or long bus fault stack frame may be generated. If an address error occurs during the exception processing for a bus error, address error, or reset, a double bus fault occurs.

8.1.4 Instruction Trap Exception

Certain instructions are used to explicitly cause trap exceptions. The TRAP #n instruction always forces an exception, and is useful for implementing system calls in user programs. The TRAPcc, TRAPV, cpTRAPcc, CHK, and CHK2 instructions force exceptions if the user program detects an error, which may be an arithmetic overflow or a subscript value that is out of bounds.

The DIVS and DIVU instructions force exceptions if a division operation is attempted with a divisor of zero. The CALLM and RTM instructions will cause a format error if an illegal privilege change is requested or invalid parameters are present in the type or option fields.

When a trap exception occurs, the processor copies the status register internally, enters the supervisor privilege level, and clears the trace bits. If tracing is enabled for the instruction that caused the trap, a trace exception is taken after the RTE instruction from the trap handler is executed, and the trace corresponds to the trap instruction; the trap handler routine is not traced. The processor generates a vector number according to the instruction being executed; for the TRAP #n instruction, the vector number is 32 plus n. The stack frame saves the trap vector offset, the program counter, and the internal copy of the status register on the supervisor stack. The saved value of the program counter is the logical address of the instruction following the instruction that caused the trap. For all instruction traps other than TRAP #n, a pointer to the instruction that caused the trap is also saved. Instruction execution resumes at the address in the exception vector after the required instruction prefetches.

8.1.5 Illegal Instruction and Unimplemented Instruction Exceptions

An illegal instruction is an instruction that contains any bit pattern in its first word that does not correspond to the bit pattern of the first word of a valid MC68020 instruction, or a MOVEC instruction with an undefined register specification field in the first extension word. An illegal instruction exception corresponds to vector number 4, and occurs when the processor attempts to execute an illegal instruction.

An illegal instruction exception is also taken if a breakpoint acknowledge bus cycle (described in **7.4.2 Breakpoint Acknowledge Cycle**) is terminated with the assertion of the bus error signal. This implies that the external circuitry did not supply an instruction word to replace the BKPT instruction word in the instruction pipe.

Instruction word patterns with bits [15–12] equal to $A are referred to as unimplemented instructions with A-line opcodes. When the processor attempts to execute an unimplemented instruction with an A-line opcode, an exception is generated with vector number 10, permitting efficient emulation of unimplemented instructions.

Instructions that have word patterns with bits [15–12] equal to $F, bits [11–9] equal to $0, and defined word patterns for subsequent words are legal PMMU instructions. Instructions that have bits [15–12] of the first words equal to $F and bits [11–9] equal to $0 but undefined patterns in the following words are treated as unimplemented instructions with F-line opcodes when execution is attempted in supervisor mode. When execution of the same instruction is attempted in user mode, a privilege violation exception is taken. The exception vector number for an unimplemented instruction with an F-line opcode is number 11.

The word patterns with bits [15–12] equal to $F and bits [11–9] not equal to zero are used for coprocessor instructions. When the processor identifies a coprocessor instruction, it runs a bus cycle referencing CPU space type $2 (refer to **4.2 ADDRESS SPACE TYPES**) and addressing one of eight coprocessors (0–7, according to bits [11–9]). If the addressed coprocessor is not included in the system and the cycle terminates with the assertion of the bus error signal, the instruction takes an unimplemented instruction (F-line opcode) exception. The system can emulate the functions of the coprocessor with an F-line exception handler. Refer to **SECTION 9 COPROCESSOR INTERFACE DESCRIPTION** for more details.

Exception processing for illegal and unimplemented instructions is similar to that for instruction traps. When the processor has identified an illegal or unimplemented instruction, it initiates exception processing instead of attempting to execute the instruction. The processor copies the status register, enters the supervisor privilege level, and clears the trace bits, disabling further tracing. The processor generates the vector number, either 4, 10, or 11, according to the exception type. The illegal or unimplemented instruction vector offset, current program counter and copy of the status register are saved on the supervisor stack, with the saved value of the program counter being the address of the illegal or unimplemented instruction. Instruction execution resumes at the address contained in the exception vector. It is the responsibility of the handling routine to adjust the stacked program counter if the instruction is emulated in software or is to be skipped on return from the handler.

8.1.6 Privilege Violation Exception

In order to provide system security, the instructions listed in Table 8-2 are privileged. An attempt to execute one of the privileged instructions while at the user privilege level causes a privilege violation exception. Also, a privilege violation exception occurs if a coprocessor requests a privilege check and the processor is at the user level.

Table 8-2. Privileged Instructions

ANDI TO SR	MOVEC
EORI to SR	MOVES
cpRESTORE	ORI to SR
cpSAVE	RESET
MOVE from SR	RTE
MOVE toSR	STOP
MOVE USP	

Exception processing for privilege violations is similar to that for illegal instructions. When the processor identifies a privilege violation, it begins exception processing before executing the instruction. The processor copies the status register, enters the supervisor privilege level, and clears the trace bits. The processor generates vector number 8, the privilege violation exception vector, and saves the privilege violation vector offset, the current program counter value, and the internal copy of the status register on the supervisor stack. The saved value of the program counter is the logical address of the first word of the instruction that caused the privilege violation. Instruction execution resumes after the required prefetches from the address in the privilege violation exception vector.

8.1.7 Trace Exception

To aid in program development, the M68000 processors include an instruction-by-instruction tracing capability. The MC68020 can be programmed to trace all instructions or only instructions that change program flow. In the trace mode, an instruction generates a trace exception after it completes execution, allowing a debugger program to monitor execution of a program.

The T1 and T0 bits in the supervisor portion of the status register control tracing. The state of these bits when an instruction begins execution determines whether the instruction generates a trace exception after the instruction completes. Clearing both T bits disables tracing, and instruction execution proceeds normally. Clearing the T1 bit and setting the T0 bit causes an instruction that forces a change of flow to take a trace exception. Instructions that increment the program counter normally do not take the trace exception. Instructions that are traced in this mode include all branches, jumps, instruction traps, returns, and coprocessor instructions that modify the program counter flow. This mode also includes status register manipulations, because the processor must re-prefetch instruction words to fill the pipe again any time an instruction that can modify the status register is executed. The execution of the BKPT instruction causes a change of flow if the opcode replacing the BKPT is an instruction that causes a change of flow (i.e., a jump, branch, etc.). Setting the T1 bit and clearing the T0 bit causes the execution of all instructions to force trace exceptions. Table 8-3 shows the trace mode selected by each combination of bits T1 and T0.

Table 8-3. Tracing Control

TI	TO	Tracing Function
0	0	No Tracing
0	1	Trace on Change of Flow (BRA, JMP, etc.)
1	0	Trace on Instruction Execution (Any Instruction)
1	1	Undefined, Reserved

In general terms, a trace exception is an extension to the function of any traced instruction. That is, the execution of a traced instruction is not complete until the trace exception processing is completed. If an instruction does not complete due to a bus error or address error exception, trace exception processing is deferred until after the execution of the suspended instruction is resumed and the instruction execution completes normally. If an interrupt is pending at the completion of an instruction, the trace exception processing occurs before the interrupt exception processing starts. If an instruction forces an exception as part of its normal execution, the forced exception processing occurs before the trace exception is processed. See **8.1.12 Multiple Exceptions** for a more complete discussion of exception priorities.

When the processor is in the trace mode and attempts to execute an illegal or unimplemented instruction, that instruction does not cause a trace exception since it is not executed. This is of particular importance to an instruction emulation routine that performs the instruction function, adjusts the stacked program counter to skip the unimplemented instruction, and returns. Before returning, the trace bits of the status register on the stack should be checked. If tracing is enabled, the trace exception processing should be emulated also, in order for the trace exception handler to account for the emulated instruction.

The exception processing for a trace starts at the end of normal processing for the traced instruction and before the start of the next instruction. The processor makes an internal copy of the status register and enters the supervisor privilege level. It also clears the T0 and T1 bits of the status register, disabling further tracing. The processor supplies vector number 9 for the trace exception and saves the trace exception vector offset, program counter value, and the copy of the status register on the supervisor stack. The saved value of the program counter is the logical address of the next instruction to be executed. Instruction execution resumes after the required prefetches from the address in the trace exception vector.

The STOP instruction does not perform its function when it is traced. A STOP instruction that begins execution with T1 = 1 and T0 = 0 forces a trace exception after it loads the status register. Upon return from the trace handler routine, execution continues with the instruction following the STOP, and the processor never enters the stopped condition.

8.1.8 Format Error Exception

Just as the processor checks that prefetched instructions are valid, the processor (with the aid of a coprocessor, if needed) also performs some checks of data values for control operations, including the type and option fields of the descriptor for CALLM, the coprocessor state frame format word for a cpRESTORE instruction, and the stack frame format for an RTE or an RTM instruction.

The RTE instruction checks the validity of the stack format code. For long bus cycle fault format frames, the RTE instruction also compares the internal version number of the processor to that contained in the frame at memory location SP + 54 (SP + $36). This check ensures that the processor can correctly interpret internal state information from the stack frame. The CALLM and RTM both check the values in the option and type fields in the module descriptor and module stack frame, respectively. If these fields do not contain proper values or if an illegal access rights change request is detected by an external memory management unit, then an illegal call or return is being requested and is not executed. Refer to **APPENDIX D.1 MODULE SUPPORT** for more information on the module call/return mechanism.

The cpRESTORE instruction passes the format word of the coprocessor state frame to the coprocessor for validation. If the coprocessor does not recognize the format value, it signals the MC68020 to take a format error exception. Refer to **SECTION 9 COPROCESSOR INTERFACE DESCRIPTION** for details of coprocessor-related exceptions.

If any of the checks previously described determine that the format of the stacked data is improper, the instruction generates a format error exception. This exception saves a short format stack frame, generates exception vector number 14, and continues execution at the address in the format exception vector. The stacked program counter value is the logical address of the instruction that detected the format error.

8.1.9 Interrupt Exceptions

When a peripheral device requires the services of the MC68020 or is ready to send information that the processor requires, it may signal the processor to take an interrupt exception. The interrupt exception transfers control to a routine that responds appropriately.

The peripheral device uses the active low interrupt priority level signals (IPL2–IPL0) to signal an interrupt condition to the processor and to specify the priority of that condition. The three signals encode a value of zero through seven (IPL0 is the least significant bit). High levels on all three signals correspond to no interrupt requested (level 0) and low levels on IPL2–IPL0 correspond to interrupt request level 7. Values one through seven specify one of seven levels of prioritized interrupts; level seven has the highest priority. External circuitry can chain or otherwise merge signals from devices at each level, allowing an unlimited number of devices to interrupt the processor.

The IPL2–IPL0 interrupt signals must maintain the interrupt request level until the MC68020 acknowledges the interrupt in order to guarantee that the interrupt is recognized. The MC68020 continuously samples the IPL2–IPL0 signals on consecutive falling edges of the processor clock in order to synchronize and debounce these signals. An interrupt request that is held constant for two consecutive clock periods is considered a valid input. Although the protocol requires that the request remain until the processor runs an interrupt acknowledge cycle for that interrupt value, an interrupt request that is held for as short a period as two clock cycles could be recognized.

The status register of the MC68020 contains an interrupt priority mask (I2, I1, I0, bits 10-8). The value in the interrupt mask is the highest priority level that the processor ignores. When an interrupt request has a priority higher than the value in the mask, the processor makes the request a pending interrupt. Figure 8-2 is a flowchart of the procedure for making an interrupt pending.

When several devices are connected to the same interrupt level, each device should hold its interrupt priority level constant until its corresponding interrupt acknowledge cycle to ensure that all requests are processed.

Table 8-4 lists the interrupt levels, the states of IPL2–IPL0 that define each level, and the mask value that allows an interrupt at each level.

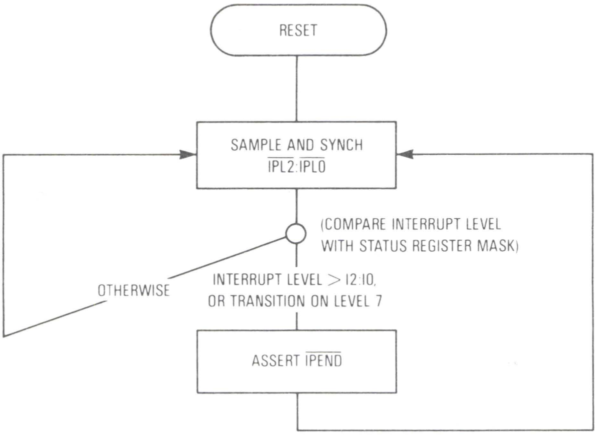

RESET
SAMPLE AND SYNCH IPL2:IPL0
(COMPARE INTERRUPT LEVEL WITH STATUS REGISTER MASK)
OTHERWISE
INTERRUPT LEVEL > I2:I0, OR TRANSITION ON LEVEL 7
ASSERT IPEND

Figure 8-2. Interrupt Pending Procedure

Table 8-4. Interrupt Levels and Mask Values

Requested Interrupt Level	Control Line Status			Interrupt Mask Level Required for Recognition
	$\overline{\text{IP2}}$	$\overline{\text{IP1}}$	$\overline{\text{IP0}}$	
0*	High	High	High	N/A*
1	High	High	Low	0
2	High	Low	High	1–0
3	High	Low	Low	2–0
4	Low	High	High	3–0
5	Low	High	Low	4–0
6	Low	Low	High	5–0
7	Low	Low	Low	7–0

*Indicates that no interrupt is requested.

Priority level seven, the nonmaskable interrupt (NMI), is a special case. Level seven interrupts cannot be masked by the interrupt priority mask, and they are transition sensitive. The processor recognizes an interrupt request each time the external interrupt request level changes from some lower level to level seven, regardless of the value in the mask. Figure 8-3 shows two examples of interrupt recognitions, one for level six and one for level seven. When the MC68020 processes a level 6 interrupt, the status register mask is automatically updated with a value of six before entering the handler routine so that subsequent level six interrupts are masked. Provided no instruction that lowers the mask value is executed, the external request can be lowered to level three and then raised back to level six, and a second level six interrupt is not processed. However, if the MC68020 is handling a level seven interrupt (status register mask set to 7) and the external request is lowered to level three and than raised back to level seven, a second level seven interrupt is processed. The second level seven interrupt is processed because the level seven interrupt is transition sensitive. A level seven interrupt is also generated by a level comparison if the request level and mask level are at seven and the priority mask is then set to a lower level (with the MOVE to SR or RTE instruction, for example). As shown in Figure 8-3 for level six interrupt request level and mask level, this is the case for all interrupt levels.

Note that a mask value of six and a mask value of seven both inhibit request levels of one through six from being recognized. In addition, neither masks a transition to an interrupt request level of seven. The only difference between mask values of six and seven occurs when the interrupt request level is seven and the mask value is seven. If the mask value is lowered to six, a second level seven interrupt is recognized.

The MC68020 asserts the interrupt pending signal ($\overline{\text{IPEND}}$) when it makes an interrupt request pending. Figure 8-4 shows the assertion of $\overline{\text{IPEND}}$ relative to the assertion of an interrupt level on the IPL lines. $\overline{\text{IPEND}}$ signals to external devices that an interrupt exception will be taken at an upcoming instruction boundary (following any higher priority exception).

The state of the $\overline{\text{IPEND}}$ signal is internally checked by the processor once per instruction, independently of bus operation. In addition, it is checked during the second instruction prefetch associated with exception processing. Figure 8-5 is a flowchart of the interrupt recognition and associated exception processing sequence.

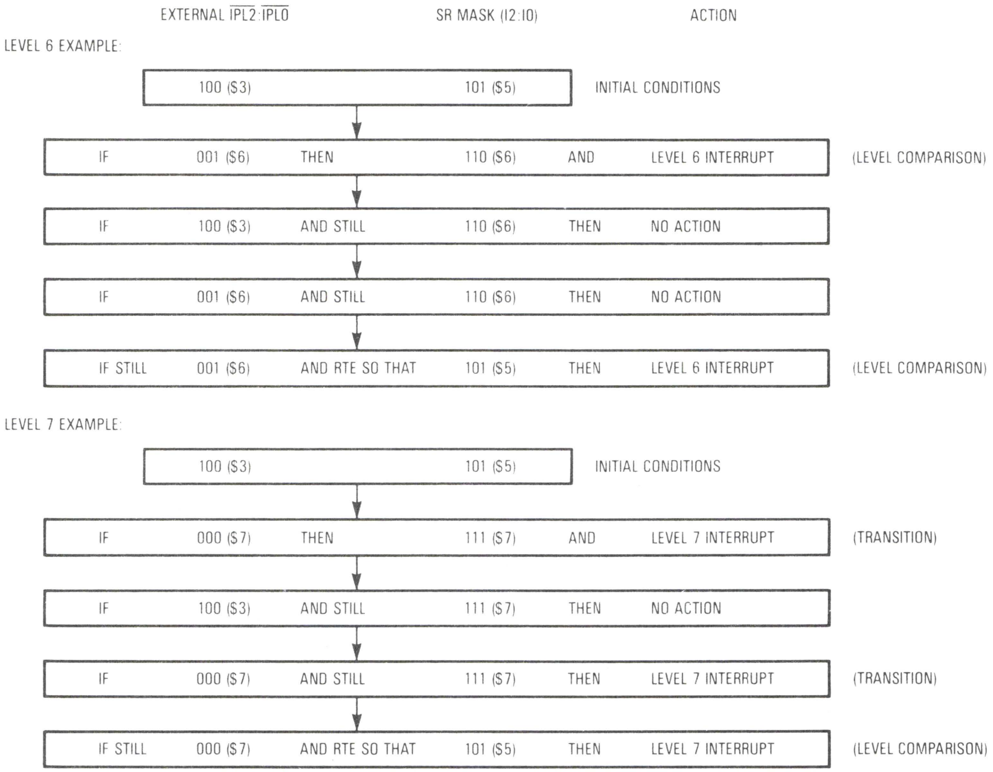

EXTERNAL $\overline{IPL2}$:$\overline{IPL0}$
SR MASK (I2:I0)
ACTION
LEVEL 6 EXAMPLE:
100 ($3) 101 ($5) INITIAL CONDITIONS
IF 001 ($6) THEN 110 ($6) AND LEVEL 6 INTERRUPT (LEVEL COMPARISON)
IF 100 ($3) AND STILL 110 ($6) THEN NO ACTION
IF 001 ($6) AND STILL 110 ($6) THEN NO ACTION
IF STILL 001 ($6) AND RTE SO THAT 101 ($5) THEN LEVEL 6 INTERRUPT (LEVEL COMPARISON)
LEVEL 7 EXAMPLE:
100 ($3) 101 ($5) INITIAL CONDITIONS
IF 000 ($7) THEN 111 ($7) AND LEVEL 7 INTERRUPT (TRANSITION)
IF 100 ($3) AND STILL 111 ($7) THEN NO ACTION
IF 000 ($7) AND STILL 111 ($7) THEN LEVEL 7 INTERRUPT (TRANSITION)
IF STILL 000 ($7) AND RTE SO THAT 101 ($5) THEN LEVEL 7 INTERRUPT (LEVEL COMPARISON)

Figure 8-3. Interrupt Recognition Examples

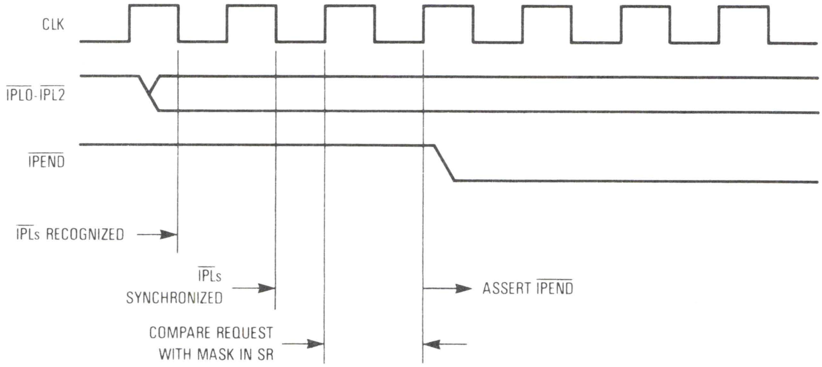

CLK
$\overline{IPL0}$-$\overline{IPL2}$
$\overline{IPEND}$
$\overline{IPL}$s RECOGNIZED
$\overline{IPL}$s SYNCHRONIZED
COMPARE REQUEST WITH MASK IN SR
ASSERT $\overline{IPEND}$

Figure 8-4. Assertion of $\overline{IPEND}$

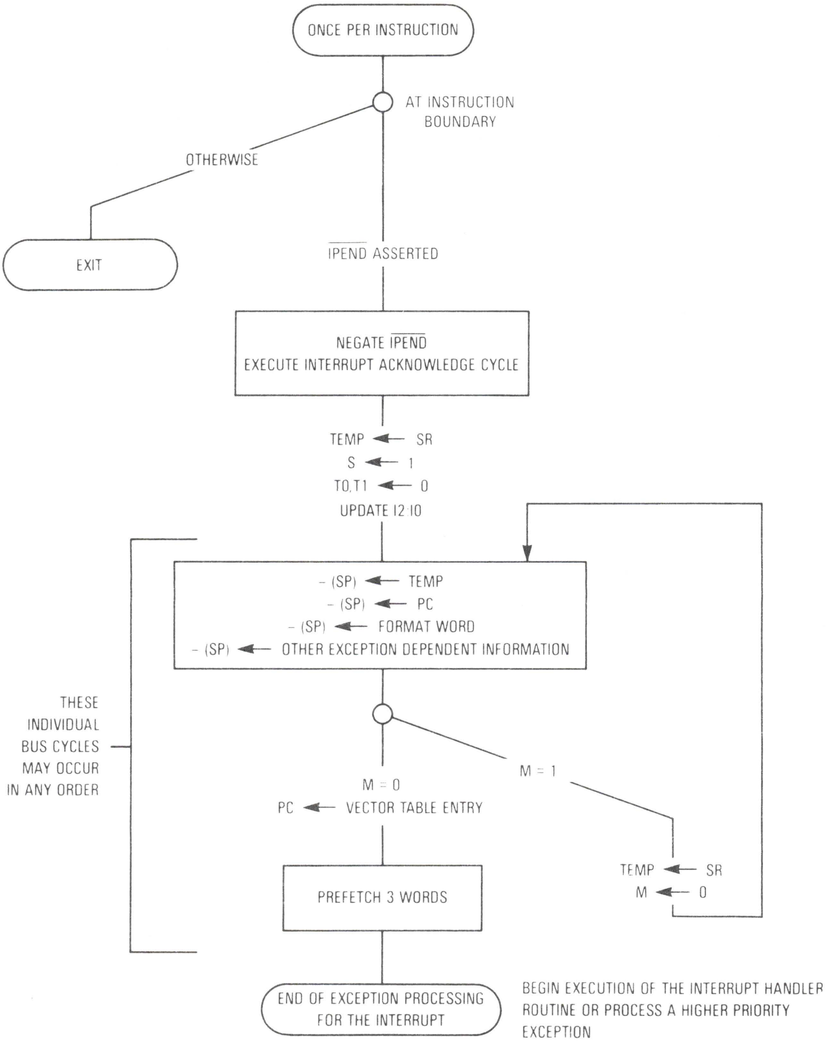

ONCE PER INSTRUCTION
AT INSTRUCTION BOUNDARY
OTHERWISE
EXIT
IPEND ASSERTED
NEGATE IPEND
EXECUTE INTERRUPT ACKNOWLEDGE CYCLE
TEMP ← SR
S ← 1
T0,T1 ← 0
UPDATE I2:I0
– (SP) ← TEMP
– (SP) ← PC
– (SP) ← FORMAT WORD
– (SP) ← OTHER EXCEPTION DEPENDENT INFORMATION
THESE INDIVIDUAL BUS CYCLES MAY OCCUR IN ANY ORDER
M = 0
PC ← VECTOR TABLE ENTRY
M = 1
TEMP ← SR
M ← 0
PREFETCH 3 WORDS
END OF EXCEPTION PROCESSING FOR THE INTERRUPT
BEGIN EXECUTION OF THE INTERRUPT HANDLER ROUTINE OR PROCESS A HIGHER PRIORITY EXCEPTION

Figure 8-5. Interrupt Exception Processing Flowchart

If no higher priority interrupt has been synchronized, the $\overline{\text{IPEND}}$ signal is negated during state 0 (S0) of an interrupt acknowledge cycle (refer to **7.4.1.1 INTERRUPT ACNOWLEDGE CYCLE — TERMINATED NORMALLY**), and at this time, the $\overline{\text{IPLx}}$ signals for the interrupt being acknowledged can be negated.

When processing an interrupt exception, the processor first makes an internal copy of the status register, sets the privilege level to supervisor, suppresses tracing, and sets the processor interrupt mask level to the level of the interrupt being serviced. The processor attempts to obtain a vector number from the interrupting device using an interrupt acknowledge bus cycle with the interrupt level number output on pins A3–A1 of the address bus. For a device that cannot supply an interrupt vector, the autovector signal ($\overline{AVEC}$) can be asserted and the MC68020 uses an internally generated autovector, which is one of vector numbers 31–25, that corresponds to the interrupt level number. If external logic indicates a bus error during the interrupt acknowledge cycle, the interrupt is considered spurious, and the processor generates the spurious interrupt vector number, 24. Refer to **7.4.1 Interrupt Acknowledge Bus Cycles** for complete interrupt bus cycle information.

Once the vector number is obtained, the processor saves the exception vector offset, program counter value, and the internal copy of the status register on the active supervisor stack. The saved value of the program counter is the logical address of the instruction that would have been executed had the interrupt not occurred. If the interrupt was acknowledged during the execution of a coprocessor instruction, further internal information is saved on the stack so that the MC68020 can continue executing the coprocessor instruction when the interrupt handler completes execution.

If the M bit of the status register is set, the processor clears the M bit and creates a throwaway exception stack frame on top of the interrupt stack as part of interrupt exception processing. This second frame contains the same program counter value and vector offset as the frame created on top of the master stack but has a format number of 1 instead of 0 or 9. The copy of the status register saved on the throwaway frame is exactly the same as that placed on the master stack except that the S bit is set in the version placed on the interrupt stack. (It may or may not be set in the copy saved on the master stack.) The resulting status register (after exception processing) has the S bit set and the M bit cleared.

The processor loads the address in the exception vector into the program counter, and normal instruction execution resumes after the required prefetches for the interrupt handler routine.

Most M68000 Family peripherals use programmable interrupt vector numbers as part of the interrupt request/acknowledge mechanism of the system. If this vector number is not initialized after reset and the peripheral must acknowledge an interrupt request, the peripheral usually returns the vector number for the uninitialized interrupt vector, 15.

8.1.10 Breakpoint Instruction Exception

In order to use the MC68020 in a hardware emulator, it must provide a means of inserting breakpoints in the emulator code and of performing appropriate operations at each breakpoint. For the MC68000 and MC68008, this can be done by inserting an illegal instruction at the breakpoint and detecting the illegal instruction exception from its vector location. However, since the vector base register on the MC68010 and MC68020 allows arbitrary relocation of exception vectors, the exception address cannot reliably identify a breakpoint. The MC68020 processor provides a breakpoint capability with a set of breakpoint instructions, $4848–$484F, for eight unique breakpoints. The breakpoint facility also allows external hardware to monitor the execution of a program residing in the on-chip instruction cache without severe performance degradation.

When the MC68020 executes a breakpoint instruction, it performs a breakpoint acknowledge cycle (read cycle) from CPU space type $0 with address lines A4–A2 corresponding to the breakpoint number. Refer to Figure 7-33 for the CPU space type $0 addresses, and **7.4.2 Breakpoint Acknowledge Cycle** for a description of the breakpoint acknowledge cycle. The external hardware can return either $\overline{\mathrm{BERR}}$ or $\overline{\mathrm{DSACKx}}$ with an instruction word on the data bus. If the bus cycle terminates with $\overline{\mathrm{BERR}}$, the processor performs illegal instruction exception processing. If the bus cycle terminates with $\overline{\mathrm{DSACKx}}$, the processor uses the data returned to replace the breakpoint instruction in the internal instruction pipe and begins execution of that instruction. The remainder of the pipe remains unaltered. In addition, no stacking or vector fetching is involved with the execution of the instruction. Figure 8-6 is a flowchart of the breakpoint instruction execution.

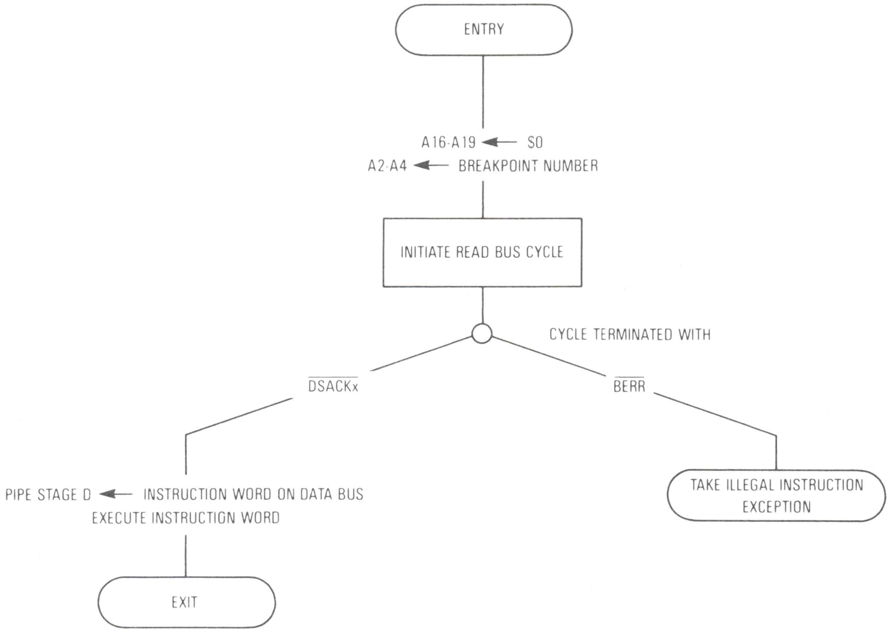

ENTRY
A16-A19 ← $0
A2-A4 ← BREAKPOINT NUMBER
INITIATE READ BUS CYCLE
CYCLE TERMINATED WITH
DSACKx
BERR
PIPE STAGE D ← INSTRUCTION WORD ON DATA BUS
EXECUTE INSTRUCTION WORD
TAKE ILLEGAL INSTRUCTION EXCEPTION
EXIT

Figure 8-6. Breakpoint Instruction Flowchart

8.1.11 Multiple Exceptions

When several exceptions occur simultaneously, they are processed according to a fixed priority. Table 8-5 lists the exceptions, grouped by characteristics. Each group has a priority, from 4–0. Priority 0 has the highest priority.

As soon as the MC68020 has completed exception processing for a condition when another exception is pending, it begins exception processing for the pending exception instead of executing the exception handler for the original exception condition. Also, whenever a bus error or address error occurs, its exception processing takes precedence over lower priority exceptions and occurs immediately. For example, if a bus error occurs during the exception processing for a trace condition, the system processes the bus error and executes its handler

Table 8-5. Exception Priority Groups

Group/ Priority	Exception and Relative Priority	Characteristics
0	0.0 — Reset	Aborts all processing (instruction or exception) and does not save old context.
1	1.0 — Address Error 1.1 — Bus Error	Suspends processing (instruction or exception) and saves internal context.
2	2.0 — BKPT #n, CALLM, CHK, CHK2, cp Midinstruction, Cp Protocol Violation, cpTRAPcc, Divide by Zero, RTE, RTM, TRAP #n, TRAPV	Exception processing is part of instruction execution.
3	3.0 — Illegal Instruction, Line A, Unimplemented Line F, Privilege Violation, cp Preinstruction	Exception processing begins before instruction is executed.
4	4.0 — cp Postinstruction 4.1 — Trace 4.2 — Interrupt	Exception processing begins when current instruction or previous exception processing is completed.

0.0 is the highest priority, 4.2 is the lowest.

before completing the trace exception processing. However, most exceptions cannot occur during exception processing, and very few combinations of the exceptions shown in Table 8-6 can be pending simultaneously.

The priority scheme is very important in determining the order in which exception handlers execute when several exceptions occur at the same time. As a general rule, the lower the priority of an exception, the sooner the handler routine for that exception executes. For example, if simultaneous trap, trace, and interrupt exceptions are pending, the exception processing for the trap occurs first, followed immediately by exception processing for the trace and then for the interrupt. When the processor resumes normal instruction execution, it is in the interrupt handler, which returns to the trace handler, which returns to the trap exception handler. This rule does not apply to the reset exception; its handler is executed first even though it has the highest priority because the reset operation clears all other exceptions.

8.1.12 Return from Exception

After the processor has completed exception processing for all pending exceptions, the processor resumes normal instruction execution at the address in the vector for the last exception processed. Once the exception handler has completed execution, the processor must return to the system context prior to the exception (if possible). The RTE instruction returns from the handler to the previous system context for any exception.

When the processor executes an RTE instruction, it examines the stack frame on top of the active supervisor stack to determine if it is a valid frame and what type of context restoration it requires. This section describes the processing for each of the stack frame types; refer to **8.3 COPROCESSOR CONSIDERATIONS** for a description of the stack frame type formats.

For a normal four-word frame, the processor updates the status register and program counter with the data read from the stack, increments the stack pointer by eight, and resumes normal instruction execution.

For the throwaway four-word stack, the processor reads the status register (SR) value from the frame, increments the active stack pointer by eight, updates the SR with the value read from the stack, and then begins RTE processing again, as shown in Figure 8-7. The processor reads a new format word from the stack frame on top of the active stack (which may or may not be the same stack used for the previous operation) and performs the proper operations corresponding to that format. In most cases, the throwaway frame is on the interrupt stack and when the SR value is read from the stack, the S and M bits are set. In that case, there is a normal four-word frame or a ten-word coprocessor midinstruction frame on the master stack. However, the second frame may be any format (even another throwaway frame) and may reside on any of the three system stacks.

For the six-word stack frame, the processor restores the status register and program counter values from the stack, increments the active supervisor stack pointer by 12, and resumes normal instruction execution.

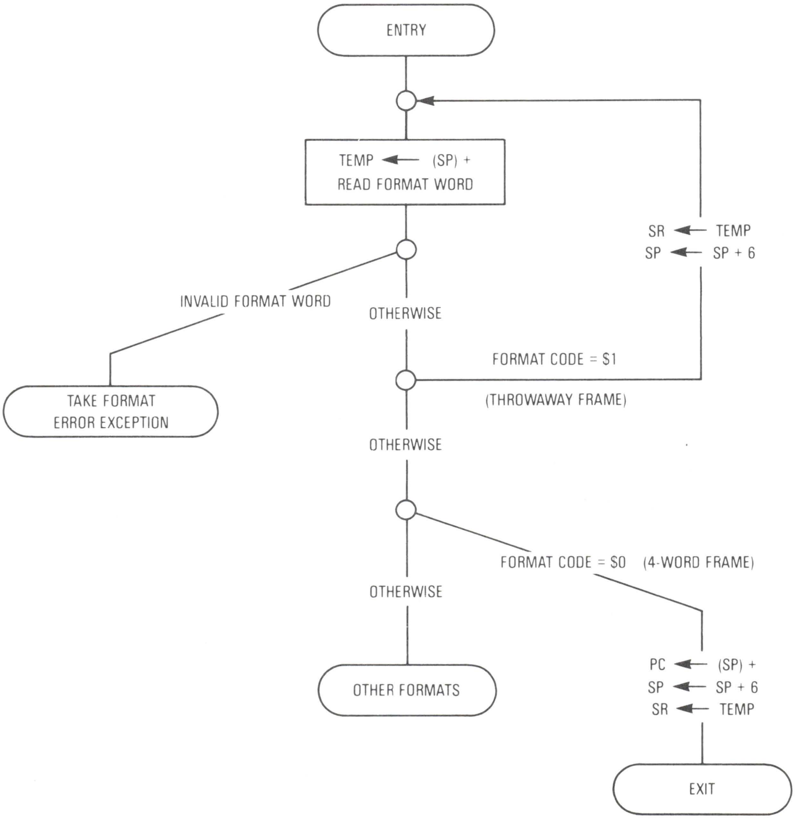

ENTRY
TEMP ← (SP) +
READ FORMAT WORD
SR ← TEMP
SP ← SP + 6
INVALID FORMAT WORD
OTHERWISE
TAKE FORMAT ERROR EXCEPTION
FORMAT CODE = $1
(THROWAWAY FRAME)
OTHERWISE
FORMAT CODE = $0 (4-WORD FRAME)
OTHERWISE
OTHER FORMATS
PC ← (SP) +
SP ← SP + 6
SR ← TEMP
EXIT

Figure 8-7. RTE Instruction for Throwaway Four-Word Frame

For the coprocessor midinstruction stack frame, the processor reads the status register, program counter, instruction address, internal register values, and the evaluated effective address from the stack, restores these values to the corresponding internal registers, and increments the stack pointer by 20. The processor then reads from the response register of the coprocessor that initiated the exception to determine the next operation to be performed. Refer to **SECTION 9 COPROCESSOR INTERFACE DESCRIPTION** for details of coprocessor related exceptions.

For both the short and long bus fault stack frames, the processor first checks the format value on the stack for validity. In addition, for the long stack frame, the processor compares the version number in the stack with its own version number. The version number is located in the most significant nibble (bits 15–12) of the word at location SP+$36 (hexadecimal) in the long stack frame. This validity check is required in a multiprocessor system to ensure that the data is properly interpreted by the RTE instruction. The RTE instruction also reads from both ends of the stack frame to make sure it is accessible. If the frame is invalid or inaccessible, the processor takes a format error or a bus error exception, respectively. Otherwise, the processor reads the entire frame into the proper internal registers, deallocates the stack, and resumes normal processing. Once the processor begins to load the frame to restore its internal state, the assertion of the $\overline{\text{BERR}}$ signal causes the processor to enter the halted state. Refer to **8.2 BUS FAULT RECOVERY** for a description of the processing that occurs after the frame is read into the internal registers.

If a format error or bus error exception occurs during the frame validation sequence of the RTE instruction, either due to any of the errors previously described or due to an illegal format code, the processor creates a normal four word or a bus cycle fault stack frame below the frame that it was attempting to use. In this way, the faulty stack frame remains intact. The exception handler can examine or repair the faulty frame. In a multiprocessor system, the faulty frame can be left to be used by another processor of a different type (for example, an MC68010 or a future M68000 processor) when appropriate.

8.2 BUS FAULT RECOVERY

An address error exception or a bus error exception indicates a bus fault. The saving of the processor state for a bus error or address error is described in **8.1.2 Bus Error Exception**, and the restoring of the processor state by an RTE instruction is described in **8.1.12 Return from Exception**.

Processor accesses of either data items or the instruction stream can result in bus errors. When a bus error exception occurs while accessing a data item, the exception is taken immediately after the bus cycle terminates. A bus error occurring during an instruction stream access is not processed until the processor attempts to use the information (if ever) that the access should have provided. For instruction faults, when the short format frame applies, the address of the pipe stage B word is the value in the program counter plus four, and the address of the stage C word is the value in the program counter plus two. For the long format, the long word at SP+$24 contains the address of the stage B word; the address of the stage C word is the address of the stage B word minus two. Address error faults occur only for instruction stream accesses, and the exceptions are taken before the bus cycles are attempted.

8.2.1 Special Status Word

The internal special status word (Figure 8-8) is one of several registers saved as part of the bus fault exception stack frame. Both the short bus cycle fault format and the long bus cycle fault format include this word at offset $A. The bus cycle fault stack frame formats are described in detail at the end of this section.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FC	FB	RC	RB	0	0	0	DF	RM	RW	SIZE		0	FC2–FC0		

FC	— Fault on Stage C of the Instruction Pipe
FB	— Fault on Stage B of the Instruction Pipe
RC	— Rerun Flag for Stage C of the Instruction Pipe*
RB	— Rerun Flag for Stage B of the Instruction Pipe*
DF	— Fault/Rerun Flag for Data Cycle*
RM	— Read-Modify-Write on Data Cycle
RW	— Read/Write for Data Cycle — 1 = Read, 0 = Write
SIZE	— Size Code for Data Cycle
FC2–FC0	— Address Space for Data Cycle

*1 = Rerun Faulted Bus Cycle or Run Pending Prefetch

0 = Do Not Rerun Bus Cycle

Figure 8-8. Special Status Word (SSW)

The special status word (SSW) information indicates whether the fault was caused by an access to the instruction stream, data stream, or both. The high-order half of the SSW contains two status bits each for the B and C stages of the instruction pipe. The fault bits (FB and FC) indicate that the processor attempted to use a stage (B or C) and found it to be marked as invalid due to a bus error on the prefetch for that stage. The fault bits can be used by a bus error handler to determine the cause(s) of a bus error exception. The rerun flag bits (RB and RC) are set to indicate that a fault occurred during a prefetch for the corresponding stage. A rerun bit is always set when the corresponding fault bit is set. The rerun bits indicate that the word in a stage of the instruction pipe is invalid, and the state of the bits can be used by a handler to repair the values in the pipe after an address error or a bus error, if necessary. If a rerun bit is set when the processor executes an RTE instruction, the processor may execute a bus cycle to prefetch the instruction word for the corresponding stage of the pipe (if it is required). If the rerun and fault bits are set for a stage of the pipe, the RTE instruction automatically reruns the prefetch cycle for that stage. The address space for the bus cycle is the program space for the privilege level indicated in the copy of the status register on the stack. If a rerun bit is cleared, the words on the stack for the corresponding stages of the pipe are accepted as valid; the processor assumes that there is no prefetch pending for the corresponding stage and that software has repaired or filled the image of the stage, if necessary.

If an address error exception occurs, the fault bits written to the stack frame are not set (they are only set due to a bus error, as previously described), and the rerun bits alone show the cause of the exception. Depending on the state of the pipeline, either RB and RC are both set, or RC alone is set. If it is desired to correct the pipeline contents and continue execution of the suspended instruction, software must place the correct instruction stream data in the stage C and/or stage B images as requested by the rerun bits and clear the rerun bits. The least significant half of the SSW applies to data cycles only. If the DF bit of the SSW is set, a data fault has occurred and caused the exception. If the DF bit is set when the processor reads the stack frame, it reruns the faulted data access; otherwise, it assumes that the data input buffer value on the stack is valid for a read or that the data has been correctly written to memory for a write (or that no data fault occurred). The RM bit of the SSW identifies a read-modify-write operation, and the RW bit indicates whether the cycle was a read or write operation. The SIZ field indicates the size of the operand access, and the FC field specifies the address space for the data cycle. Data and instruction stream faults may be pending simultaneously; the fault handler should be able to recognize any combination of the FC, FB, RC, RB, and DF bits.

8.2.2 Using Software to Complete the Bus Cycles

One method of completing a faulted bus cycle is to use a software handler to emulate the cycle. This is the only method for correcting address errors. The handler should emulate the faulted bus cycle in a manner that is transparent to the instruction that caused the fault. For instruction stream faults, the handler may need to run bus cycles for both the B and C stages of the instruction pipe. The RB and RC bits identify the stages that may require a bus cycle; the FB and FC bits indicate that a stage was found to be invalid when an attempt was made to use its contents. Those stages must be repaired. For each faulted stage, the software handler should copy the instruction word from the proper address space as indicated by the S bit of the copy of the status register saved on the stack to the image of the appropriate stage in the stack frame. In addition, the handler must clear the rerun bit associated with the stage that it has corrected. The handler should not change the fault bits FB and FC.

In order to repair data faults (indicated by DF = 1), the software should first examine the RM bit in the SSW to determine if the fault was generated during a read-modify-write operation. If RM = 0, the handler should then check the R/W bit of the SSW to determine if the fault was caused by a read or a write cycle. For data write faults, the handler must transfer the properly sized data from the data output buffer (DOB) on the stack frame to the location indicated by the data fault address in the address space defined by the SSW. (Both the DOB and the data fault address are part of the stack frame, at SP + \$18 and SP + \$10, respectively.) Data read faults only generate the long bus fault frame and the handler must transfer properly sized data from the location indicated by the fault address and address space to the image of the data input buffer (DIB) at location SP + \$2C of the long format stack frame. Byte, word, and 3-byte operands are right-justified in the 4-byte data buffers. In addition, the software handler must clear the DF bit of the SSW to indicate that the faulted bus cycle has been corrected.

In order to emulate a read-modify-write cycle, the exception handler must first read the operation word at the program counter address (SP + 2 of the stack frame). This word identifies the CAS, CAS2, or TAS instruction that caused the fault. Then the handler must emulate this entire instruction (which may consist of up to four long-word transfers) and update the condition code portion of the status register appropriately, because the RTE instruction expects the entire operation to have been completed if the RM bit (in SSW) is set and the DF bit (in SSW) is cleared. This is true even if the fault occurred on the first read cycle.

To emulate the entire instruction, the handler must save the data and address registers for the instruction (with a MOVEM instruction, for example). Next, the handler reads and modifies (if necessary) the memory location. It clears the DF bit in the SSW of the stack frame, modifies the condition codes in the status register copy, and the copies of any data or address registers required for the CAS and CAS2 instructions. Last, the handler restores the registers that it saved at the beginning of the emulation. Except for the data input buffer (DIB), the copy of the status register, and the special status word, the handler should not modify a bus fault stack frame. The only bits in the special status word that may be modified are DF, RB, and RC; all other bits, including those defined for internal use, must remain unchanged.

Address error faults must be repaired in software. Address error faults can be distinguished from bus error faults by the value in the vector offset field of the format word.

8.2.3 Completing the Bus Cycles with RTE

Another method of completing a faulted bus cycle is to allow the processor to rerun the bus cycles during execution of the RTE instruction that terminates the exception handler. This method cannot be used to recover from address errors. The RTE instruction is always executed. Unless the handler routine has corrected the error and cleared the fault (and cleared the rerun and DF bits of the special status word) the RTE instruction cannot complete the bus cycle(s). If the DF bit is still set at the time of the RTE execution, the faulted data cycle is rerun by the RTE instruction. If the fault bit for a stage of the pipe is set and the corresponding rerun bit was not cleared by the software, the RTE reruns the associated instruction prefetch. The fault occurs again unless the cause of the fault, such as a non-resident page in a virtual memory system, has been corrected. If the rerun bit is set for a stage of the pipe and the fault bit is cleared, the associated prefetch cycle may or may not be run by the RTE instruction (depending on whether the stage is required).

If a fault occurs when the RTE instruction attempts to rerun the bus cycle(s), the processor creates a new stack frame on the supervisor stack after deallocating the previous frame, and address error or bus error exception processing starts in the normal manner.

The read-modify-write operations of the MC68020 can also be completed by the RTE instruction that terminates the handler routine. The rerun operation executed by the RTE instruction with the DF bit of the SSW set reruns the entire instruction. If the cause of the error has been corrected, the handler does not need to emulate the instruction but can leave the DF bit set and execute the RTE instruction.

Systems programmers and designers should both be aware that the memory management unit of the MC68020 treats any bus cycle with $\overline{\text{RMC}}$ asserted as a write operation for protection checking regardless of the state of R/$\overline{\text{W}}$ signal. Otherwise, the potential for partially destroying system pointers with CAS and CAS2 instructions exists since one portion of the write operation could take place and the remainder be aborted by a bus error.

8.3 COPROCESSOR CONSIDERATIONS

Exception handler programmers should consider carefully whether to save and restore the context of a coprocessor at the beginning and end of handler routines for exceptions that can occur during the execution of a coprocessor instruction (i.e., bus errors, interrupts, and coprocessor-related exceptions). The nature of the coprocessor and the exception handler routine determines whether or not saving the state of one or more coprocessors with the cpSAVE and cpRESTORE instructions is required. If the coprocessor allows multiple coprocessor instructions to be executed concurrently, it may require its state to be saved and restored for all coprocessor-generated exceptions regardless of whether or not the coprocessor is accessed during the handler routine. The MC68882 floating-point coprocessor is an example of this type of coprocessor. The MC68881 floating-point coprocessor, on the other hand, requires FSAVE and FRESTORE instructions within an exception handler routine only if the exception handler itself uses the coprocessor.

8.4 EXCEPTION STACK FRAME FORMATS

The MC68020 provides six different stack frames for exception processing. The set of frames includes the normal four- and six-word stack frames, the four-word throwaway stack frame, the coprocessor midinstruction exception stack frame, and the short and long bus fault stack frames.

When the MC68020 writes or reads a stack frame, it uses long-word operand transfers wherever possible. Using a long-word-aligned stack pointer with memory that is on a 32-bit port greatly enhances exception processing performance. The processor does not necessarily read or write the stack frame data in sequential order.

The system software should not depend on a particular exception generating a particular stack frame. For compatibility with future devices, the software should be able to handle any type of stack frame for any type of exception.

Table 8-6 summarizes the stack frames defined for the M68000 Family.

Table 8-6. Exception Stack Frames (Sheet 1 of 2)

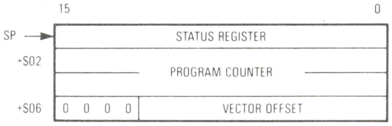
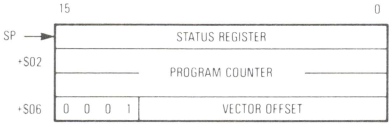
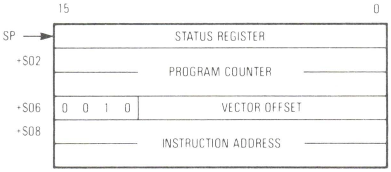
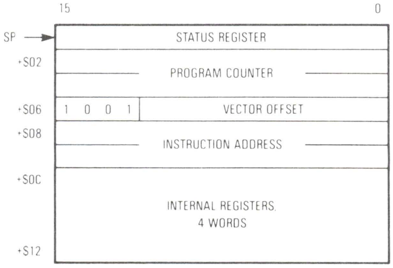
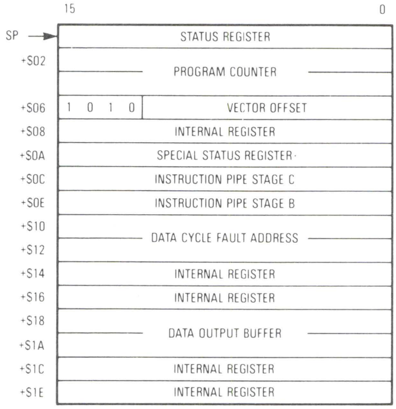
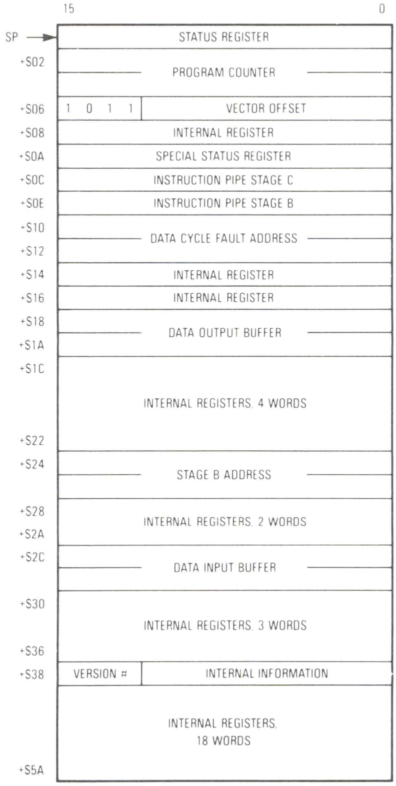
Stack Frames	Exception Types (Stacked PC Points to)
 15 0 SP +$02 +$06 STATUS REGISTER PROGRAM COUNTER 0 0 0 0 VECTOR OFFSET FOUR WORD STACK FRAME - FORMAT $0	● Interrupt [Next instruction] ● Format Error [RTE or cpRESTORE instruction] ● TRAP #N [Next instruction] ● Illegal Instruction [Illegal instruction] ● A-Line Instruction [A-line instruction] ● F-Line Instruction [F-line instruction] ● Privilege Violation [First word of instruction causing Privilege Violation] ● Coprocessor Pre-Instruction [Op-Word of instruction that returned the Take Pre-Instruction primitive]
 15 0 SP +$02 +$06 STATUS REGISTER PROGRAM COUNTER 0 0 0 1 VECTOR OFFSET THROWAWAY FOUR WORD STACK FRAME - FORMAT $1	● Created on Interrupt Stack during interrupt exception processing when transition from master state to interrupt state occurs [Next instruction — same as on master stack]
 15 0 SP +$02 +$06 +$08 STATUS REGISTER PROGRAM COUNTER 0 0 1 0 VECTOR OFFSET INSTRUCTION ADDRESS SIX WORD STACK FRAME - FORMAT $2	● CHK ● CHK2 ● cpTRAPcc ● TRAPcc ● TRAPV ● Trace ● Zero Divide ● MMU Configuration ● Coprocessor Post-Instruction [Next instruction for all these exceptions] INSTRUCTION ADDRESS is the address of the instruction tht caused the exception
 15 0 SP +$02 +$06 +$08 +$0C +$12 STATUS REGISTER PROGRAM COUNTER 1 0 0 1 VECTOR OFFSET INSTRUCTION ADDRESS INTERNAL REGISTERS, 4 WORDS COPROCESSOR MID-INSTRUCTION STACK FRAME (10 WORDS) - FORMAT $9	● Coprocessor Mid-Instruction ● Main-Detected Protocol Violation ● Interrupt Detected During Coprocessor Instruction (supported with 'null come again with interrupts allowed' primitive) [Next word to be fetched from instruction stream for all these exceptions] INSTRUCTION ADDRESS is the address of the instruction that caused the exception

Table 8-6. Exception Stack Frames (Sheet 2 of 2)

Stack Frames	Exception Types (Stacked PC Points to)
 15 0 SP +$02 +$06 +$08 +$0A +$0C +$0E +$10 +$12 +$14 +$16 +$18 +$1A +$1C +$1E STATUS REGISTER PROGRAM COUNTER 1 0 1 0 VECTOR OFFSET INTERNAL REGISTER SPECIAL STATUS REGISTER INSTRUCTION PIPE STAGE C INSTRUCTION PIPE STAGE B DATA CYCLE FAULT ADDRESS INTERNAL REGISTER INTERNAL REGISTER DATA OUTPUT BUFFER INTERNAL REGISTER INTERNAL REGISTER SHORT BUS CYCLE FAULT STACK FRAME (16 WORDS) - FORMAT $A	• Address Error or Bus Error — Execution Unit at Instruction Boundary [Next instruction]
 15 0 SP +$02 +$06 +$08 +$0A +$0C +$0E +$10 +$12 +$14 +$16 +$18 +$1A +$1C +$22 +$24 +$28 +$2A +$2C +$30 +$36 +$38 +$5A STATUS REGISTER PROGRAM COUNTER 1 0 1 1 VECTOR OFFSET INTERNAL REGISTER SPECIAL STATUS REGISTER INSTRUCTION PIPE STAGE C INSTRUCTION PIPE STAGE B DATA CYCLE FAULT ADDRESS INTERNAL REGISTER INTERNAL REGISTER DATA OUTPUT BUFFER INTERNAL REGISTERS, 4 WORDS STAGE B ADDRESS INTERNAL REGISTERS, 2 WORDS DATA INPUT BUFFER INTERNAL REGISTERS, 3 WORDS VERSION # INTERNAL INFORMATION INTERNAL REGISTERS, 18 WORDS LONG BUS CYCLE FAULT STACK FRAME (46 WORDS) - FORMAT $B	• Address Error or Bus Error — Instruction Execution in Progress [Address of instruction in execution when fault occurred — may not be the instruction that generated the faulted bus cycle]

SECTION 9
COPROCESSOR INTERFACE DESCRIPTION

The M68000 Family of general-purpose microprocessors provides a level of performance that satisfies a wide range of computer applications. Special-purpose hardware, however, can often provide a higher level of performance for a specific application. The coprocessor concept allows the capabilities and performance of a general-purpose processor to be enhanced for a particular application without encumbering the main processor architecture. A coprocessor can efficiently meet specific capability requirements that must typically be implemented in software by a general-purpose processor. With a general-purpose main processor and the appropriate coprocessor(s), the processing capabilities of a system can be tailored to a specific application.

The MC68020 supports the M68000 coprocessor interface described in this section. The section is intended for designers who are implementing coprocessors to interface with the MC68020.

The designer of a system that uses one or more Motorola coprocessors (the MC68881 or MC68882 floating-point coprocessor, for example) does not require a detailed knowledge of the M68000 coprocessor interface. Motorola coprocessors conform to the interface described in this section. Typically, they implement a subset of the interface, and that subset is described in the coprocessor user's manual. These coprocessors execute Motorola defined instructions that are described in the user's manual for each coprocessor.

9.1 INTRODUCTION

The distinction between standard peripheral hardware and a M68000 coprocessor is important from a perspective of the programming model. The programming model of the main processor consists of the instruction set, register set, and memory map available to the programmer. An M68000 coprocessor is a device or set of devices that communicates with the main processor through the protocol defined as the M68000 coprocessor interface. The programming model for a coprocessor is different than that for a peripheral device. A coprocessor adds additional instructions and generally additional registers and data types to the programming model that are not directly supported by the main processor architecture. The additional instructions are dedicated coprocessor instructions that utilize the coprocessor capabilities. The necessary interactions between the main processor and the coprocessor that provide a given service are transparent to the programmer. That is, the programmer does not need to know the specific communication protocol between the main processor and the coprocessor because this protocol is implemented in hardware. Thus, the coprocessor can provide capabilities to the user without appearing separate from the main processor.

In contrast, standard peripheral hardware is generally accessed through interface registers mapped into the memory space of the main processor. To use the services provided by the peripheral, the programmer accesses the peripheral registers with standard processor instructions. While a peripheral could conceivably provide capabilities equivalent to a coprocessor for many applications, the programmer must implement the communication protocol between the main processor and the peripheral necessary to use the peripheral hardware.

The communication protocol defined for the M68000 coprocessor interface is described in **9.2 COPROCESSOR INSTRUCTION TYPES**. The algorithms that implement the M68000 coprocessor interface are provided in the microcode of the MC68020 and are completely transparent to the MC68020 programmer's model. For example, floating-point operations are not implemented in the MC68020 hardware. In a system utilizing both the MC68020 and the MC68881 or MC68882 floating-point coprocessor, a programmer can use any of the instructions defined for the coprocessor without knowing that the actual computation is performed by the MC68881 or MC68882 hardware.

9.1.1 Interface Features

The M68000 coprocessor interface design incorporates a number of flexible capabilities. The physical coprocessor interface uses the main processor external bus, which simplifies the interface since no special-purpose signals are involved. With the MC68020, a coprocessor uses the asynchronous bus transfer protocol. Since standard bus cycles transfer information between the main processor and the coprocessor, the coprocessor can be implemented in whatever technology is available to the coprocessor designer. A coprocessor can be implemented as a VLSI device, as a separate system board, or even as a separate computer system.

Since the main processor and a M68000 coprocessor can communicate using the asynchronous bus, they can operate at different clock frequencies. The system designer can choose the speeds of a main processor and coprocessor that provide the optimum performance for a given system. Both the MC68881 and MC68882 floating-point coprocessors use the asynchronous bus handshake protocol.

The M68000 coprocessor interface also facilitates the design of coprocessors. The coprocessor designer must only conform to the coprocessor interface, and does not need an extensive knowledge of the architecture of the main processor. Also, the main processor can operate with a coprocessor without having explicit provisions made in the main processor for the capabilities of that coprocessor. This provides a great deal of freedom in the implementation of a given coprocessor.

9.1.2 Concurrent Operation Support

The programmer's model for the M68000 Family of microprocessors is based on sequential, nonconcurrent instruction execution. This implies that the instructions in a given sequence must appear to be executed in the order in which they occur. In order to maintain a uniform programmer's model, any coprocessor extensions should also maintain the model of sequential, nonconcurrent instruction execution at the user level. Consequently, the programmer can assume that the images of registers and memory affected by a given instruction have been updated when the next instruction in the sequence accessing these registers or memory locations is executed.

The M68000 coprocessor interface provides full support of all operations necessary for nonconcurrent operation of the main processor and its associated coprocessors. Although the M68000 coprocessor interface allows concurrency in coprocessor execution, the coprocessor designer is responsible for implementing this concurrency while maintaining a programming model based on sequential nonconcurrent instruction execution.

For example, if the coprocessor determines that instruction ''B'' does not use or alter resources to be altered or used by instruction ''A'', instruction ''B'' can be executed concurrently (if the execution hardware is also available). Thus, the required instruction interdependencies and sequences of the program are always respected. The MC68882 coprocessor offers concurrent instruction execution while the MC68881 coprocessor does not. However, the MC68020 can execute instructions concurrently with coprocessor instruction execution in the MC68881.

9.1.3 Coprocessor Instruction Format

The instruction set for a given coprocessor is defined by the design of that coprocessor. When a coprocessor instruction is encountered in the main processor instruction stream, the MC68020 hardware initiates communication with the coprocessor and coordinates any interaction necessary to execute the instruction with the coprocessor. A programmer needs to know only the instruction set and register set defined by the coprocessor in order to use the functions provided by the coprocessor hardware.

The instruction set of an M68000 coprocessor uses a subset of the F-line operation words in the M68000 instruction set. The operation word is the first word of any M68000 Family instruction. The F-line operation word contains ones in bits 15–12 ([15–12] = 1111; refer to Figure 9-1); the remaining bits are coprocessor and instruction dependent. The F-line operation word may be followed by as many extension words as are required to provide additional information necessary for the execution of the coprocessor instruction.

As shown in Figure 9-1, bits 11–9 of the F-line operation word encode the coprocessor identification code (CpID). The MC68020 uses the coprocessor identification field to indicate the coprocessor to which the instruction applies. F-line operation words, in which the Cp-ID is zero, are not coprocessor instructions for the MC68020. If the Cp-ID (bits 11–9) is 000, the instruction accesses the MC68851 paged memory management unit. Instructions with a Cp-ID of zero and a nonzero type field are unimplemented instructions that cause the MC68020 to begin exception processing. The MC68020 never generates coprocessor interface bus cycles with the Cp-ID equal to zero (except via the MOVES instruction).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Cp-ID			TYPE			TYPE DEPENDENT					

Figure 9-1. F-line Coprocessor Instruction Operation Word

Cp-ID codes of 000-101 are reserved for current and future Motorola coprocessors, and Cp-ID codes of 110-111 are reserved for user-defined coprocessors. The Motorola Cp-ID codes that are currently defined are: 000 for the MC68851 paged-memory management unit and 001 for the MC68881 or MC68882 floating-point coprocessor. By default, Motorola assemblers will use Cp-ID codes 000 and 001 when generating the instruction operation codes for the MC68881, MC68882, or MC68851 coprocessor instructions.

The encoding of bits 8–0 of the coprocessor instruction operation word is dependent on the particular instruction being implemented (refer to **9.2 COPROCESSOR INSTRUCTION TYPES**).

9.1.4 Coprocessor System Interface

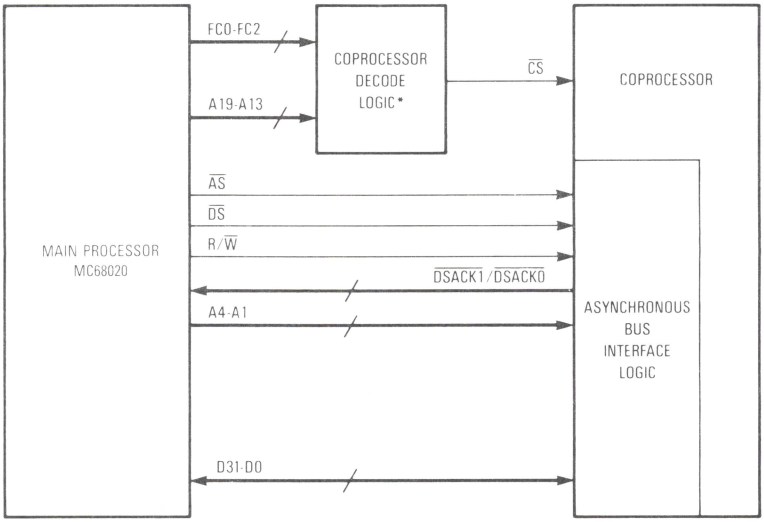
The communication protocol between the main processor and coprocessor necessary to execute a coprocessor instruction uses a group of interface registers called coprocessor interface registers resident within the coprocessor. By accessing one of these interface register, the MC68020 hardware initiates coprocessor instructions. The coprocessor uses a set of response primitive codes and format codes defined for the M68000 coprocessor interface to communicate status and service requests to the main processor through these registers. The coprocessor interface registers are also used to pass operands between the main processor and the coprocessor. The coprocessor interface register set, response primitives, and format codes are discussed in **9.3 COPROCESSOR INTERFACE REGISTER SET** and **9.4 COPROCESSOR RESPONSE PRIMITIVES**.

9.1.4.1 COPROCESSOR CLASSIFICATION. M68000 coprocessors can be classified into two categories depending on their bus interface capabilities. The first category, non-DMA coprocessors, consists of coprocessors that always operate as bus slaves. The second category, DMA coprocessors, consists of coprocessors that operate as bus slaves while communicating with the main processor across the coprocessor interface. These coprocessors also have the ability to operate as bus masters and directly control the system bus.

If the operation of a coprocessor does not require a large portion of the available bus bandwidth or has special requirements not directly satisfied by the main processor, that coprocessor can be efficiently implemented as a non-DMA coprocessor. Since non-DMA coprocessors always operate as bus slaves, all external bus-related functions that the coprocessor requires are performed by the main processor. The main processor transfers operands from the coprocessor by reading the operand from the appropriate coprocessor interface register (CIR) and then writing the operand to a specified effective address with the appropriate address space specified on the function code lines. Likewise, the main processor transfers operands to the coprocessor by reading the operand from a specified effective address (and address space) and then writing that operand to the appropriate CIR using the coprocessor interface. The bus interface circuitry of a coprocessor operating as a bus slave is not as complex as that of a device operating as a bus master.

To improve the efficiency of operand transfers between memory and the coprocessor, a coprocessor that requires a relatively high amount of bus bandwidth or has special bus requirements can be implemented as a DMA coprocessor. DMA coprocessors can operate as bus masters. The coprocessor provides all control, address, and data signals necessary to request and obtain the bus, and then performs DMA transfers using the bus. DMA coprocessors, however, must still act as bus slaves when they require information or services of the main processor using the M68000 coprocessor interface protocol.

9.1.4.2 PROCESSOR-COPROCESSOR INTERFACE. Figure 9-2 is a block diagram of the signals involved in an asynchronous non-DMA M68000 coprocessor interface. Since the Cp-ID on signals A15–A13 of the address bus is used with other address signals to select the coprocessor, the system designer can use several coprocessors of the same type and assign a unique Cp-ID to each one.


MAIN PROCESSOR MC68020
FC0-FC2
A19-A13
COPROCESSOR DECODE LOGIC*
$\overline{CS}$
COPROCESSOR
$\overline{AS}$
$\overline{DS}$
R/$\overline{W}$
$\overline{DSACK1}$/$\overline{DSACK0}$
A4-A1
ASYNCHRONOUS BUS INTERFACE LOGIC
D31-D0

FC0-FC2 = 111 → CPU SPACE CYCLE
A19-A16 = 0010 → COPROCESSOR ACCESS IN CPU SPACE
A15-A13 = xxx → COPROCESSOR IDENTIFICATION
A4-A1 = rrrr → COPROCESSOR INTERFACE REGISTER SELECTOR

*Chip select logic may be integrated into the coprocessor
Address lines not specified above are "0" during coprocessor access

Figure 9-2. Asynchronous Non-DMA M68000 Coprocessor Interface Signal Usage

The MC68851 paged memory management unit, however, uses FC2–FC0 and A31–A0 as inputs to perform address translations. Thus, the MC68851 decodes the information on the MC68020 address bus and function code lines internally to determine when an access is made to one of its coprocessor interface registers. Since the MC68851 decodes the address and function code pins intenally, it requires no external chip select logic and thus must be assigned the Cp-ID 000.

The MC68020 accesses the registers in the coprocessor interface register set using standard asynchronous bus cycles. Thus, the bus interface implemented by a coprocessor for its interface register set must satisfy the MC68020 address, data, and control signal timing. The MC68020 timing information for read and write cycles is illustrated in Figures 12-5–12-8 on foldout pages in the back of this manual. The MC68020 bus operation is described in detail in **SECTION 7 BUS OPERATION**.

During coprocessor instruction execution, the MC68020 executes CPU space bus cycles to access the interface register set of a coprocessor. The MC68020 drives the three function code outputs high (FC2:FC0 = 111) identifying a CPU space bus cycle. The coprocessor interface register set is mapped into CPU space in the same manner that a peripheral interface register set is generally mapped into data space. The information encoded on the function code lines and address bus of the MC68020 during a coprocessor access is used to generate the chip select signal for the coprocessor being accessed. Other address lines select a register within the interface set. The information encoded on the function code and address lines of the MC68020 during a coprocessor access is illustrated in Figure 9-3.

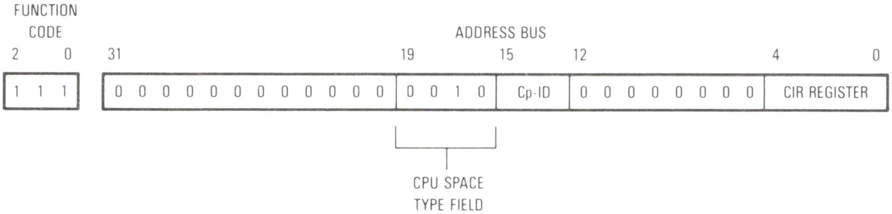


Figure 9-3. MC68020 CPU Space Address Encodings

Address signals A19–A16 specify the CPU space cycle type for a CPU space bus cycle. The types of CPU space cycles currently defined for the MC68020 are interrupt acknowledge, breakpoint acknowledge, module support operations, and coprocessor access cycles. CPU space type $2 (A19–A16 = 0010) specifies a coprocessor access cycle.

Signals A15–A13 of the MC68020 address bus specify the coprocessor identification code (Cp-ID) for the coprocessor being accessed. This code is transferred from bits 11–9 of the coprocessor instruction operation word (refer to Figure 9-1) to the address bus during each coprocessor access. Thus, decoding the MC68020 function code signals and bits A19–A13 of the address bus provides a unique chip select signal for a given coprocessor. The function code signals and A19–A16 indicate a coprocessor access; A15–A13 indicate which of the possible eight coprocessors (000 through 111) is being accessed. Bits A31–A20 and A12–A5 of the MC68020 address bus are always zero during a coprocessor access.

The MC68010 can emulate coprocessor access cycles in CPU space using the MOVES instruction.

9.1.4.3 COPROCESSOR INTERFACE REGISTER SELECTION. Figure 9-4 shows that the value on the MC68020 address bus during a coprocessor access addresses a unique region of the main processor's CPU address space. Signals A4–A0 of the MC68020 address bus select the CIR being accessed. The register map for the M68000 coprocessor interface is shown in Figure 9-5. The individual registers are described in detail in **9.3 COPROCESSOR INTERFACE REGISTER SET**.

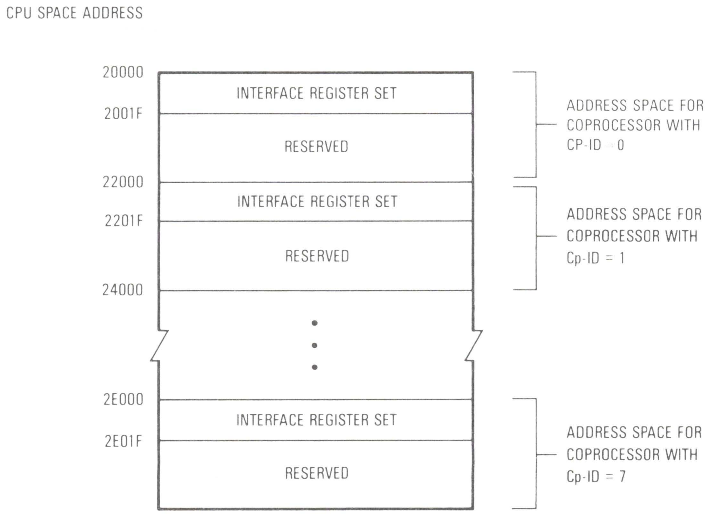

CPU SPACE ADDRESS
20000
2001F
22000
2201F
24000
2E000
2E01F
INTERFACE REGISTER SET
RESERVED
INTERFACE REGISTER SET
RESERVED
INTERFACE REGISTER SET
RESERVED
ADDRESS SPACE FOR COPROCESSOR WITH CP-ID = 0
ADDRESS SPACE FOR COPROCESSOR WITH Cp-ID = 1
ADDRESS SPACE FOR COPROCESSOR WITH Cp-ID = 7

Figure 9-4. Coprocessor Address Map in MC68020 CPU Space

	31	15 0
00	RESPONSE*	CONTROL*
04	SAVE*	RESTORE*
08	OPERATION WORD	COMMAND*
0C	(RESERVED)	CONDITION*
10	OPERAND*	
14	REGISTER SELECT	(RESERVED)
18	INSTRUCTION ADDRESS	
1C	OPERAND ADDRESS	

Figure 9-5. Coprocessor Interface Register Set Map

9.2 COPROCESSOR INSTRUCTION TYPES

The M68000 coprocessor interface supports four categories of coprocessor instructions: general, conditional, context save, and context restore. The category name indicates the type of operations provided by the coprocessor instructions in the category. The instruction category also determines the CIR accessed by the MC68020 to initiate instruction and communication protocols between the main processor and the coprocessor necessary for instruction execution.

During the execution of instructions in the general or conditional categories, the coprocessor uses the set of coprocessor response primitive codes defined for the MC68000 coprocessor interface to request services from and indicate status to the main processor. During the execution of the instructions in the context save and context restore categories, the coprocessor uses the set of coprocessor format codes defined for the M68000 coprocessor interface to indicate its status to the main processor.

9.2.1 Coprocessor General Instructions

The general coprocessor instruction category contains data processing instructions and other general-purpose instructions for a given coprocessor.

9.2.1.1 FORMAT. Figure 9-6 shows the format of a general type instruction.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Cp-ID			0	0	0	EFFECTIVE ADDRESS					
COPROCESSOR COMMAND															
OPTIONAL EFFECTIVE ADDRESS OR COPROCESSOR-DEFINED EXTENSION WORDS															

Figure 9-6. Coprocessor General Instruction Format (cpGEN)

The mnemonic cpGEN is a generic mnemonic used in this discussion for all general instructions. The mnemonic of a specific general instruction usually suggests the type of operation it performs and the coprocessor to which it applies. The actual mnemonic and syntax used to represent a coprocessor instruction is determined by the syntax of the assembler or compiler that generates the object code.

A coprocessor general type instruction consists of at least two words. The first word of the instruction is an F-line operation code (bits [15–12] = 1111). The Cp-ID field of the F-line operation code is used during the coprocessor access to indicate which of the coprocessors in the system executes the instruction. During accesses to the coprocessor interface registers (refer to **9.1.4.2 PROCESSOR-COPROCESSOR INTERFACE**), the processor places the Cp-ID on address lines A15–A13.

Bits [8–6] = 000 indicate that the instruction is in the general instruction category. Bits 5–0 of the F-line operation code sometimes encodes a standard M68000 effective address specifier (refer to **2.5 EFFECTIVE ADDRESS ENCODING SUMMARY**). During the execution of a cpGEN instruction, the coprocessor can use a coprocessor response primitive to request that the MC68020 perform an effective address calculation necessary for that instruction. Using the effective address specifier field of the F-line operation code, the processor then determines the effective addressing mode. If a coprocessor never requests effective address calculation, bits 5–0 can have any value (don't cares).

The second word of the general type instruction is the coprocessor command word. The main processor writes this command word to the command CIR to initiate execution of the instruction by the coprocessor.

An instruction in the coprocessor general instruction category optionally includes a number of extension words following the coprocessor command word. These words can provide additional information required for the coprocessor instruction. For example, if the coprocessor requests that the MC68020 calculate an effective address during coprocessor instruction execution, information required for the calculation must be included in the instruction format as effective address extension words.

9.2.1.2 PROTOCOL. The execution of a cpGEN instruction follows the protocol shown in Figure 9-7. The main processor initiates communication with the coprocessor by writing the instruction command word to the command CIR. The coprocessor decodes the command word to begin processing the cpGEN instruction. Coprocessor design determines the interpretation of the coprocessor command word; the MC68020 does not attempt to decode it.

MAIN PROCESSOR		COPROCESSOR
M1 RECOGNIZE COPROCESSOR INSTRUCTION F-LINE OPERATION WORD		
M2 WRITE COPROCESSOR COMMAND WORD TO COMMAND CIR	→	C1 DECODE COMMAND WORD AND INITIATE COMMAND EXECUTION
		C2 WHILE (MAIN PROCESSOR SERVICE IS REQUIRED) DO STEPS 1) AND 2) BELOW
M3 READ COPROCESSOR RESPONSE PRIMITIVE CODE FROM RESPONSE CIR 1) PERFORM SERVICE REQUESTED BY RESPONSE PRIMITIVE 2) IF (COPROCESSOR RESPONSE PRIMITIVE INDICATES "COME AGAIN") GO TO M3 (SEE NOTE 1)	↔	1) REQUEST SERVICE BY PLACING APPROPRIATE RESPONSE PRIMITIVE CODE IN RESPONSE CIR 2) RECEIVE SERVICE FROM MAIN PROCESSOR
		C3 REFLECT "NO COME AGAIN" IN RESPONSE CIR
		C4 COMPLETE COMMAND EXECUTION
		C5 REFLECT "PROCESSING FINISHED" STATUS IN RESPONSE CIR
M4 PROCEED WITH EXECUTION OF NEXT INSTRUCTION (SEE NOTE 2)		

NOTE: 1. "Come Again" indicates that further service of the main processor is being requested by the coprocessor.
2. The next instruction should be the operatio word pointed to by the ScanPC at this point. The operation of the MC68020 ScanPC is discussed in 9.4.1 ScanPC.

Figure 9-7. Coprocessor Interface Protocol for General Category Instructions

While the coprocessor is executing an instruction, it requests any required services from and communicates status to the main processor by placing coprocessor response primitive codes in the response CIR. After writing to the command CIR, the main processor reads the response CIR and responds appropriately. When the coprocessor has completed the execution of an instruction or no longer needs the services of the main processor to execute the instruction, it provides a response to release the processor. The main processor can then execute the next instruction in the instruction stream. However, if a trace exception is pending, the MC68020 does not terminate communication with the coprocessor until the coprocessor indicates that it has completed all processing associated with the cpGEN instruction (refer to **9.5.2.5 TRACE EXCEPTIONS**).

The coprocessor interface protocol shown in Figure 9-7 allows the coprocessor to define the operation of each general category instruction. That is, the main processor initiates the instruction execution by writing the instruction command word to the command CIR and by reading the response CIR to determine its next action. The execution of the coprocessor instruction is then defined by the internal operation of the coprocessor and by its use of response primitives to request services from the main processor. This instruction protocol allows a wide range of operations to be implemented in the general instruction category.

9.2.2 Coprocessor Conditional Instructions

The conditional instruction category provides program control based on the operations of the coprocessor. The coprocessor evaluates a condition and returns a true/false indicator to the main processor. The main processor completes the execution of the instruction based on this true/false condition indicator.

The implementation of instructions in the conditional category promotes efficient use of both the main processor's and the coprocessor's hardware. The condition specified for the instruction is related to the coprocessor operation and is, therefore, evaluated by the coprocessor. The instruction completion following the condition evaluation is, however, directly related to the operation of the main processor. The main processor performs the change of flow, the setting of a byte, or the TRAP operation, since its architecture explicitly implements these operations for its instruction set.

Figure 9-8 shows the protocol for a conditional category coprocessor instruction. The main processor initiates execution of an instruction in this category by writing a condition selector to the condition CIR. The coprocessor decodes the condition selector to determine the condition to evaluate. The coprocessor can use response primitives to request that the main processor provide services required for the condition evaluation. After evaluating the condition, the coprocessor returns a true/false indicator to the main processor by placing a Null primitive (refer to **9.4.4 Null Primitive**) in the response CIR. The main processor completes the coprocessor instruction execution when it receives the condition indicator from the coprocessor.

9.2.2.1 BRANCH-ON COPROCESSOR CONDITION INSTRUCTION. The conditional instruction category includes two formats of the M68000 Family branch instruction. These instructions branch on conditions related to the coprocessor operation. They execute similarly to the conditional branch instructions provided in the M68000 Family instruction set.

9.2.2.1.1 Format. Figure 9-9 shows the format of the branch on coprocessor condition instruction that provides a word-length displacement. Figure 9-10 shows the format of the instruction that includes a long-word displacement.

MAIN PROCESSOR		COPROCESSOR
M1 RECOGNIZE COPROCESSOR INSTRUCTION F-LINE OPERATION WORD		
M2 WRITE COPROCESSOR CONDITION SELECTOR TO CONDITION CIR	→	C1 DECODE CONDITION SELECTOR AND INITIATE CONDITION EVALUATION
M3 READ COPROCESSOR RESPONSE PRIMITIVE CODE FROM RESPONSE CIR 1) PERFORM SERVICE REQUESTED BY RESPONSE PRIMITIVE 2) IF (COPROCESSOR RESPONSE PRIMITIVE INDICATES "COME AGAIN") GO TO M3 (SEE NOTE 1)	↔	C2 WHILE (MAIN PROCESSOR SERVICE IS REQUIRED DO STEPS 1) AND 2) BELOW 1) REQUEST SERVICE BY PLACING APPROPRIATE RESPONSE PRIMITIVE CODE IN RESPONSE CIR 2) RECEIVE SERVICE FROM MAIN PROCESSOR
		C3 COMPLETE CONDITION EVALUATION
		C4 REFLECT "NO COME AGAIN" STATUS WITH TRUE/FALSE CONDITION INDICATOR IN RESPONSE CIR
M4 COMPLETE EXECUTION OF INSTRUCTION BASED ON THE TRUE/FALSE CONDITION INDICATOR RETURNED IN THE RESPONSE CIR		

NOTES: 1. All coprocessor response primitives, except the Null primitive, that allows the "Come Again" primitive attribute must indicate "Come Again" when used during the execution of a conditional category instruction. If a "Come Again" attribute is not indicated in one of these primitives, the main processor will initiate protocol violation exception processing (see 9.5.2.1 PROTOCOL VIOLATIONS).

Figure 9-8. Coprocessor Interface Protocol for Conditional Category Instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Cp-ID			0	1	0	CONDITION SELECTOR					
OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS															
DISPLACEMENT															

Figure 9-9. Branch-On Coprocessor Condition Instruction (cpBcc.W)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Cp-ID			0	1	1	CONDITION SELECTOR					
OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS															
DISPLACEMENT — HIGH															
DISPLACEMENT — LOW															

Figure 9-10. Branch-On Coprocessor Condition Instruction (cpBcc.L)

The first word of the branch-on coprocessor condition instruction is the F-line operation word. Bits [15–12] = 1111 and bits [11–9] contain the identification code of the coprocessor that is to evaluate the condition. The value in bits [8–6] identifies either the word or the long-word displacement format of the branch instruction, which is specified by the cpBcc.W or cpBcc.L mnemonic, respectively.

Bits [5–0] of the F-line operation word contain the coprocessor condition selector field. The MC68020 writes the entire operation word to the condition CIR to initiate execution of the branch instruction by the coprocessor. The coprocessor uses bits [5–0] to determine which condition to evaluate.

If the coprocessor requires additional information to evaluate the condition, the branch instruction format can include this information in extension words. Following the F-line operation word, the number of extension words is determined by the coprocessor design. The final word(s) of the cpBcc instruction format contains the displacement used by the main processor to calculate the destination address when the branch is taken.

9.2.2.1.2 Protocol. Figure 9-8 shows the protocol for the cpBcc.L and cpBcc.W instructions. The main processor initiates the instruction by writing the F-line operation word to the condition CIR to transfer the condition selector to the coprocessor. The main processor then reads the response CIR to determine its next action. The coprocessor can return a response primitive to request services necessary to evaluate the condition. If the coprocessor returns the false condition indicator, the main processor executes the next instruction in the instruction stream. If the coprocessor returns the true condition indicator, the processor adds the displacement to the MC68020 scanPC (refer to **9.4.1 ScanPC**) to determine the address of the next instruction for the main processor to execute. The scanPC must be pointing to the location of the first word of the displacement in the instruction stream when the address is calculated. The displacement is a twos complement integer that can be either a 16-bit word or a 32-bit long word. The processor sign-extends the 16-bit displacement to a long word value for the destination address calculation.

9.2.2.2 SET-ON COPROCESSOR CONDITION INSTRUCTION. The set-on coprocessor condition instructions set or reset a flag (a data alterable byte) according to a condition evaluated by the coprocessor. The operation of this instruction is similar to the operation of the Scc instruction in the M68000 Family instruction set. Although the Scc instruction and the cpScc instruction do not explicitly cause a change of program flow, they are often used to set flags that control program flow.

9.2.2.2.1 Format. Figure 9-11 shows the format of the set on coprocessor condition instruction, denoted by the cpScc mnemonic.

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="3">Cp-ID</td><td>0</td><td>0</td><td>1</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="10"></td><td colspan="6">CONDITION SELECTOR</td></tr>
<tr><td colspan="16">OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS</td></tr>
<tr><td colspan="16">OPTIONAL EFFECTIVE ADDRESS EXTENSION WORDS (0-5 WORDS)</td></tr>
</table>

Figure 9-11. Set-On Coprocessor Condition (cpScc)

The first word of the cpScc instruction is the F-line operation word. This word contains the Cp-ID field in bits [11–9] and 001 in bits [8–6] to identify the cpScc instruction. The lower six bits of the F-line operation word are used to encode an M68000 Family effective addressing mode (refer to **2.5 EFFECTIVE ADDRESS ENCODING SUMMARY**).

The second word of the cpScc instruction format contains the coprocessor condition selector in bits [5–0]. Bits [15–6] of this word are reserved by Motorola and should be zero to ensure compatibility with future M68000 products. This word is written to the condition CIR to initiate the cpScc instruction.

If the coprocessor requires additional information to evaluate the condition, the instruction can include extension words to provide this information. The number of these extension words, which follow the word containing the coprocessor condition selector field, is determined by the coprocessor design.

The final portion of the cpScc instruction format contains zero to five effective address extension words. These words contain any additional information required to calculate the effective address specified by bits [5–0] of the F-line operation word.

9.2.2.2.2 Protocol. Figure 9-8 shows the protocol for the cpScc instruction. The MC68020 transfers the condition selector to the coprocessor by writing the word following the F-line operation word to the condition CIR. The main processor then reads the response CIR to determine its next action. The coprocessor can return a response primitive to request services necessary to evaluate the condition. The operation of the cpScc instruction depends on the condition evaluation indicator returned to the main processor by the coprocessor. When the coprocessor returns the false condition indicator, the main processor evaluates the effective address specified by bits [5–0] of the F-line operation word and sets the byte at that effective address to FALSE (all bits cleared). When the coprocessor returns the true condition indicator, the main processor sets the byte at the effective address to TRUE (all bits set to one).

9.2.2.3 TEST COPROCESSOR CONDITION, DECREMENT AND BRANCH INSTRUCTION. The operation of the test coprocessor condition, decrement and branch instruction is similar to that of the DBcc instruction provided in the M68000 Family instruction set. This operation uses a coprocessor-evaluated condition and a loop counter in the main processor. It is useful for implementing DO-UNTIL constructs used in many high-level languages.

9.2.2.3.1 Format. Figure 9-12 shows the format of the test coprocessor condition, decrement and branch instruction, denoted by the cpDBcc mnemonic.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Cp-ID			0	0	1	0	0	1	EFFECTIVE ADDRESS		
(RESERVED)										CONDITION SELECTOR					
OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS															
DISPLACEMENT															

Figure 9-12. Test Coprocessor Condition, Decrement and Branch Instruction Format (cpDBcc)

The first word of the cpDBcc instruction is the F-line operation word. This word contains the CP-ID field in bits [11–9] and 001001 in bits [8–3] to identify the cpDBcc instruction. Bits [2–0] of this operation word specify the main processor data register used as the loop counter during the execution of the instruction.

The second word of the cpDBcc instruction format contains the coprocessor condition selector in bits [5–0] and should contain zeros in bits [15–6] to maintain compatibility with future M68000 products. This word is written to the condition CIR to initiate execution of the cpDBcc instruction by the coprocessor.

If the coprocessor requires additional information to evaluate the condition, the cpDBcc instruction can include this information in extension words. These extension words follow the word containing the coprocessor condition selector field in the cpDBcc instruction format.

The last word of the instruction contains the displacement for the cpDBcc instruction. This displacement is a twos complement 16-bit value that is sign extended to long word size when it is used in a destination address calculation.

9.2.2.3.2 Protocol. Figure 9-8 shows the protocol for the cpDBcc instructions. The MC68020 transfers the condition selector to the coprocessor by writing the word following the operation word to the condition CIR. The main processor then reads the response CIR to determine its next action. The coprocessor can use a response primitive to request any services necessary to evaluate the condition. If the coprocessor returns the true condition indicator, the main processor executes the next instruction in the instruction stream. If the coprocessor returns the false condition indicator, the main processor decrements the low-order word of the register specified by bits [2–0] of the F-line operation word. If this register contains minus one (−1) after being decremented, the main processor executes the next instruction in the instruction stream. If the register does not contain minus one (−1) after being decremented, the main processor branches to the destination address to continue instruction execution.

The MC68020 adds the displacement to the scanPC (refer to **9.4.1 ScanPC**) to determine the address of the next instruction. The scanPC must point to the 16-bit displacement in the instruction stream when the destination address is calculated.

9.2.2.4 TRAP-ON COPROCESSOR CONDITION. The trap-on coprocessor condition instruction allows the programmer to initiate exception processing based on conditions related to the coprocessor operation.

9.2.2.4.1 Format. Figure 9-13 shows the format of the trap-on coprocessor condition instruction, denoted by the cpTRAPcc mnemonic.

The F-line operation word contains the CP-ID field in bits [11–9] and 001111 in bits [8–3] to identify the cpTRAPcc instruction. Bits [2–0] of the cpTRAPcc F-line operation word specify the number of optional operand words in the instruction format. The instruction format can include zero, one, or two operand words.

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="3">Cp-ID</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="3">OP-MODE</td></tr>
<tr><td colspan="10">(RESERVED)</td><td colspan="6">CONDITION SELECTOR</td></tr>
<tr><td colspan="16">OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS</td></tr>
<tr><td colspan="16">OPTIONAL WORD</td></tr>
<tr><td colspan="16">OR LONG-WORD OPERAND</td></tr>
</table>

Figure 9-13. Trap-On Coprocessor Condition (cpTRAPcc)

The second word of the cpTRAPcc instruction format contains the coprocessor condition selector in bits [5–0] and should contain zeros in bits [15–6] to maintain compatibility with future M68000 products. This word is written to the condition CIR of the coprocessor to initiate execution of the cpTRAPcc instruction by the coprocessor.

If the coprocessor requires additional information to evaluate a condition, the instruction can include this information in extension words. These extension words follow the word containing the coprocessor condition selector field in the cpTRAPcc instruction format.

The operand words of the cpTRAPcc F-line operation word follow the coprocessor-defined extension words. These operand words are not explicitly used by the MC68020, but can be used to contain information referenced by the cpTRAPcc exception handling routines. The valid encodings for bits [2–0] of the F-line operation word and the corresponding numbers of operand words are listed in Table 9-1. Other encodings of these bits are invalid for the cpTRAPcc instruction.

Table 9-1. cpTRAPcc Op-Mode Encodings

Op-Mode	Optional Words in Instruction Format
010	One
011	Two
100	Zero

9.2.2.4.2 Protocol. Figure 9-8 shows the protocol for the cpTRAPcc instructions. The MC68020 transfers the condition selector to the coprocessor by writing the word following the operation word to the condition CIR. The main processor then reads the response CIR to determine its next action. Using a response primitive, the coprocessor can request any services necessary to evaluate the condition. If the coprocessor returns the true condition indicator, the main processor initiates exception processing for the cpTRAPcc exception (refer to **9.5.2.4 cpTRAPcc INSTRUCTION TRAPS**). If the coprocessor returns the false condition indicator, the main processor executes the next instruction in the instruction stream.

9.2.3 Coprocessor Save and Restore Instructions

The coprocessor context save and context restore instruction categories in the M68000 coprocessor interface support multi-tasking programming environments. In a multi-tasking environment, the context of a coprocessor may need to be changed asynchronously with respect to the operation of that coprocessor. That is, the coprocessor may be interrupted at any point in the execution of an instruction in the general or conditional category in order to begin context change operations.

In contrast to the general and conditional instruction categories, the context save and context restore instruction categories do not use the coprocessor response primitives. A set of format codes defined by the M68000 coprocessor interface communicates status information to the main processor during the execution of these instructions. These coprocessor format codes are discussed in detail in **9.2.3.2 COPROCESSOR FORMAT WORDS**.

9.2.3.1 COPROCESSOR INTERNAL STATE FRAMES. The context save (cpSAVE) and context restore (cpRESTORE) instructions transfer an internal coprocessor state frame between memory and a coprocessor. This internal coprocessor state frame represents the state of coprocessor operations. Using the cpSAVE and cpRESTORE instructions, it is possible to interrupt coprocessor operation, save the context associated with the current operation, and initiate coprocessor operations with a new context.

A cpSAVE instruction stores a coprocessor's internal state frame as a sequence of long-word entries in memory. Figure 9-14 shows the format of a coprocessor state frame.

During execution of the cpSAVE instruction, the MC68020 calculates the state frame effective address from information in the operation word of the instruction and stores a format word at this effective address. The processor writes the long words that form the coprocessor state frame to descending memory addresses beginning with the address specified by the sum of the effective address and the format word length field multiplied

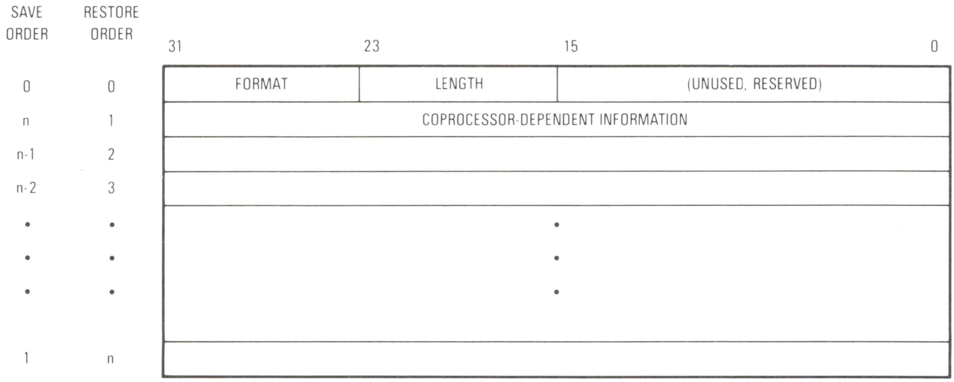


Figure 9-14. Coprocessor State Frame Format in Memory

by four. During execution of the cpRESTORE instruction, the MC68020 reads the format word and long words in the state frame from ascending addresses beginning with the effective address specified in the instruction operation word.

The processor stores the coprocessor format word at the lowest address of the state frame in memory, and this word is the first word transferred for both the cpSAVE and the cpRESTORE instructions. The word following the format word does not contain information relevant to the coprocessor state frame, but serves to keep the information in the state frame a multiple of four bytes in size. The number of entries following the format word (at higher addresses) is determined by the format word length for a given coprocessor state.

The information in a coprocessor state frame describes a context of operation for that coprocessor. This description of a coprocessor context includes the program invisible state information and, optionally, the program visible state information. The program invisible state information consists of any internal registers or status information that cannot be accessed by the program but is necessary for the coprocessor to continue its operation at the point of suspension. Program visible state information includes the contents of all registers that appear in the coprocessor programming model and that can be directly accessed using the coprocessor instruction set. The information saved by the cpSAVE instruction must include the program invisible state information. If cpGEN instructions are provided to save the program visible state of the coprocessor, the cpSAVE and cpRESTORE instructions should only transfer the program invisible state information to minimize interrupt latency during a save or restore operation.

9.2.3.2 COPROCESSOR FORMAT WORDS. The coprocessor communicates status information to the main processor during the execution of cpSAVE and cpRESTORE instructions using coprocessor format words. The format words defined for the M68000 coprocessor interface are listed in Table 9-2.

Table 9-2. Coprocessor Format Word Encodings

Format Code	Length	Meaning
00	xx	Empty Reset
01	xx	Not Ready, Come Again
02	xx	Invalid Format
03-oF	xx	Undefined, Reserved
10-FF	Length	Valid Format, Coprocessor Defined

The upper byte of the coprocessor format word contains the code used to communicate coprocessor status information to the main processor. The MC68020 recognizes four types of format words: empty/reset, not ready, invalid format, and valid format. The MC68020 interprets the reserved format codes ($03-$0F) as invalid format words. The lower byte of the coprocessor format word specifies the size in bytes (which must be a multiple of four) of the coprocessor state frame. This value is only relevant when the code byte contains the valid format code (refer to **9.2.3.2.4 Valid Format Word**).

9.2.3.2.1 Empty/Reset Format Word. The coprocessor returns the empty/reset format code during a cpSAVE instruction to indicate that the coprocessor contains no user-specific information. That is, no coprocessor instructions have been executed since either a previous cpRESTORE of an empty/reset format code or the previous hardware reset. If the main processor reads the empty/reset format word from the save CIR during the initiation of a cpSAVE instruction, it stores the format word at the effective address specified in the cpSAVE instruction, and executes the next instruction.

When the main processor reads the empty/reset format word from memory during the execution of the cpRESTORE instruction, it writes the format word to the restore CIR. The main processor then reads the restore CIR and, if the coprocessor returns the empty/reset format word, executes the next instruction. The main processor can initialize the coprocessor by writing the empty/reset format code to the restore CIR. When the coprocessor receives the empty/reset format code, it terminates any current operations and waits for the main processor to initiate the next coprocessor instruction. In particular, after the cpRESTORE of the empty/reset format word, the execution of a cpSAVE should cause the empty/reset format word to be returned when a cpSAVE instruction is executed before any other coprocessor instructions. Thus, an empty/reset state frame consists only of the format word and the following reserved word in memory (refer to Figure 9-14).

9.2.3.2.2 Not-Ready Format Word. When the main processor initiates a cpSAVE instruction by reading the save CIR, the coprocessor can delay the save operation by returning a not-ready format word. The main processor then services any pending interrupts and reads the save CIR again. The not-ready format word delays the save operation until the coprocessor is ready to save its internal state. The cpSAVE instruction can suspend execution of a general or conditional coprocessor instruction; the coprocessor can resume execution of the suspended instruction when the appropriate state is restored with a cpRESTORE. If no further main processor services are required to complete coprocessor instruction execution, it may be more efficient to complete the instruction and thus reduce the size of the saved state. The coprocessor designer should consider the efficiency of completing the instruction or of suspending and later resuming the instruction when the main processor executes a cpSAVE instruction.

When the main processor initiates a cpRESTORE instruction by writing a format word to the restore CIR, the coprocessor should usually terminate any current operations and restore the state frame supplied by the main processor. Thus, the not-ready format word should usually not be returned by the coprocessor during the execution of a cpRESTORE instruction. If the coprocessor must delay the cpRESTORE operation for any reason, it can return the not-ready format word when the main processor reads the restore CIR. If the main processor reads the not-ready format word from the restore CIR during the cpRESTORE instruction, it reads the restore CIR again without servicing any pending interrupts.

9.2.3.2.3 Invalid Format Word. When the format word placed in the restore CIR to initiate a cpRESTORE instruction does not describe a valid coprocessor state frame, the coprocessor returns the invalid format word in the restore CIR. When the main processor reads this format word during the cpRESTORE instruction, it writes the abort mask to the control CIR and initiates format error exception processing. The two least significant bits of the abort mask are 01; the fourteen most significant bits are undefined.

A coprocessor should usually not place an invalid format word in the save CIR when the main processor initiates a cpSAVE instruction. A coprocessor, however, may not be able to support the initiation of a cpSAVE instruction while it is executing a previously initiated cpSAVE or cpRESTORE instruction. In this situation, the coprocessor can return the invalid format word when the main processor reads the save CIR to initiate the cpSAVE instruction while either another cpSAVE or cpRESTORE instruction is executing. If the main processor reads an invalid format word from the save CIR it writes the abort mask to the control CIR and initiates format error exception processing (refer to **9.5.1.5 FORMAT ERRORS**).

9.2.3.2.4 Valid Format Word. When the main processor reads a valid format word from the save CIR during the cpSAVE instruction, it uses the length field to determine the size of the coprocessor state frame to save. The length field in the lower eight bits of a format word is relevant only in a valid format word. During the cpRESTORE instruction, the main processor uses the length field in the format word read from the effective address in the instruction to determine the size of the coprocessor state frame to restore.

The length field of a valid format word, representing the size of the coprocessor state frame, must contain a multiple of four. If the main processor detects a value that is not a multiple of four in a length field during the execution of a cpSAVE or cpRESTORE instruction, the main processor writes the abort mask (refer to **9.2.3.2.3 Invalid Format Word**) to the control CIR and initiates format error exception processing.

9.2.3.3 COPROCESSOR CONTEXT SAVE INSTRUCTION. The M68000 coprocessor context save instruction category consists of one instruction. The coprocessor context save instruction, denoted by the cpSAVE mnemonic, saves the context of a coprocessor dynamically without relation to the execution of coprocessor instructions in the general or conditional instruction categories. During the execution of a cpSAVE instruction, the coprocessor communicates status information to the main processor by using the coprocessor format codes.

9.2.3.3.1 Format. Figure 9-15 shows the format of the cpSAVE instruction. The first word of the instruction is the F-line operation word, which contains the coprocessor identification code in bits [11–9] and an M68000 effective address code in bits [5–0]. The effective address encoded in the cpSAVE instruction is the address at which the state frame associated with the current context of the coprocessor is saved in memory.

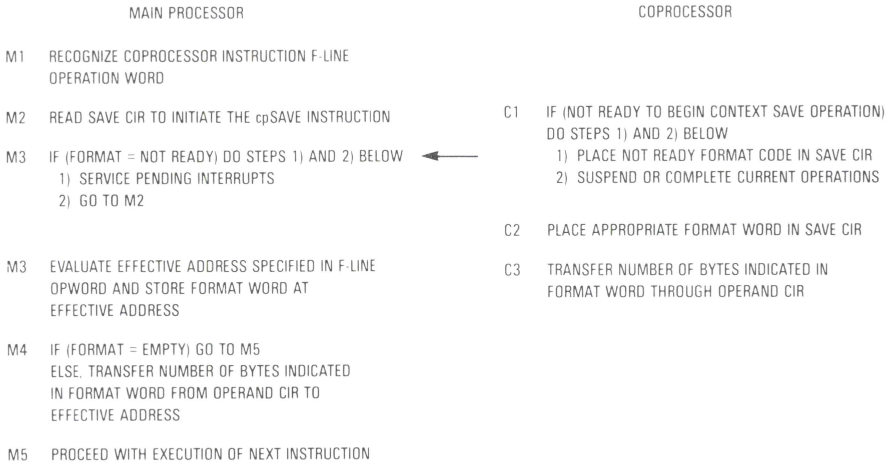
The control alterable and predecrement addressing modes are valid for the cpSAVE instruction. Other addressing modes cause the MC68020 to initiate F-line emulator exception processing as described in **9.5.2.2 F-LINE EMULATOR EXCEPTIONS**.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Cp-ID			1	0	0	EFFECTIVE ADDRESS					
EFFECTIVE ADDRESS EXTENSION WORDS (0-5 WORDS)															

Figure 9-15. Coprocessor Context Save Instruction Format (cpSAVE)

The instruction can include as many as five effective address extension words following the cpSAVE instruction operation word. These words contain any additional information required to calculate the effective address specified by bits [5–0] of the operation word.

9.2.3.3.2 Protocol. Figure 9-16 shows the protocol for the coprocessor context save instruction. The main processor initiates execution of the cpSAVE instruction by reading the save CIR. Thus, the cpSAVE instruction is the only coprocessor instruction that begins by reading from a CIR. (All other coprocessor instructions write to a CIR to initiate execution of the instruction by the coprocessor.) The coprocessor communicates status information associated with the context save operation to the main processor by placing coprocessor format codes in the save CIR.



MAIN PROCESSOR

M1 RECOGNIZE COPROCESSOR INSTRUCTION F-LINE OPERATION WORD

M2 READ SAVE CIR TO INITIATE THE cpSAVE INSTRUCTION

M3 IF (FORMAT = NOT READY) DO STEPS 1) AND 2) BELOW
  1) SERVICE PENDING INTERRUPTS
  2) GO TO M2

M3 EVALUATE EFFECTIVE ADDRESS SPECIFIED IN F-LINE OPWORD AND STORE FORMAT WORD AT EFFECTIVE ADDRESS

M4 IF (FORMAT = EMPTY) GO TO M5
ELSE, TRANSFER NUMBER OF BYTES INDICATED IN FORMAT WORD FROM OPERAND CIR TO EFFECTIVE ADDRESS

M5 PROCEED WITH EXECUTION OF NEXT INSTRUCTION

COPROCESSOR

C1 IF (NOT READY TO BEGIN CONTEXT SAVE OPERATION) DO STEPS 1) AND 2) BELOW
  1) PLACE NOT READY FORMAT CODE IN SAVE CIR
  2) SUSPEND OR COMPLETE CURRENT OPERATIONS

C2 PLACE APPROPRIATE FORMAT WORD IN SAVE CIR

C3 TRANSFER NUMBER OF BYTES INDICATED IN FORMAT WORD THROUGH OPERAND CIR

Figure 9-16. Coprocessor Context Save Instruction Protocol

If the coprocessor is not ready to suspend its current operation when the main processor reads the save CIR, it returns a not-ready format code. The main processor services any pending interrupts and then reads the save CIR again. After placing the not-ready format code in the save CIR, the coprocessor should either suspend or complete the instruction it is currently executing.

Once the coprocessor has suspended or completed the instruction it is executing, it places a format code representing the internal coprocessor state in the save CIR. When the main processor reads the save CIR, it transfers the format word to the effective address specified in the cpSAVE instruction. The lower byte of the coprocessor format word specifies the number of bytes of state information, not including the format word and associated null word, to be transferred from the coprocessor to the effective address specified. If the state information is not a multiple of four bytes in size, the MC68020 initiates format error exception processing (refer to **9.5.1.5 FORMAT ERRORS**). The coprocessor and main processor coordinate the transfer of the internal state of the coprocessor using the operand CIR. The MC68020 completes the coprocessor context save by repeatedly reading the operand CIR and writing the information obtained into memory until all the bytes specified in the coprocessor format word have been transferred. Following a cpSAVE instruction, the coprocessor should be in an idle state, that is, not executing any coprocessor instructions.

The cpSAVE instruction is a privileged instruction. When the main processor identifies a cpSAVE instruction, it checks the supervisor bit in the status register to determine whether it is operating at the supervisor privilege level. If the MC68020 attempts to execute a cpSAVE instruction while at the user privilege level (status register bit [13]=0), it initiates privilege violation exception processing without accessing any of the coprocessor interface registers (refer to **9.5.2.3 PRIVILEGE VIOLATIONS**).

The MC68020 initiates format error exception processing if it reads an invalid format word (or a valid format word whose length field is not a multiple of four bytes) from the save CIR during the execution of a cpSAVE instruction (refer to **9.2.3.2.3 Invalid Format Word**). The MC68020 writes an abort mask (refer to **9.2.3.2.3 Invalid Format Word**) to the control CIR to abort the coprocessor instruction prior to beginning exception processing. Figure 9-16 does not include this case since a coprocessor usually returns either a not ready or a valid format code in the context of the cpSAVE instruction. The coprocessor can return the invalid format word, however, if a cpSAVE is initiated while the coprocessor is executing a cpSAVE or cpRESTORE instruction and the coprocessor is unable to support the suspension of these two instructions.

9.2.3.4 COPROCESSOR CONTEXT RESTORE INSTRUCTION. The M68000 coprocessor context restore instruction category includes one instruction. The coprocessor context restore instruction, denoted by the cpRESTORE mnemonic, forces a coprocessor to terminate any current operations and to restore a former state. During the execution of a cpRESTORE instruction, the coprocessor can communicate status information to the main processor by placing format codes in the restore CIR.

9.2.3.4.1 Format. Figure 9-17 shows the format of the cpRESTORE instruction.

The first word of the instruction is the F-line operation word, which contains the coprocessor identification code in bits [11–9] and an M68000 effective addressing code in bits [5–0]. The effective address encoded in the cpRESTORE instruction is the starting address in memory where the coprocessor context is stored. The effective address is that of the coprocessor format word that applies to the context to be restored to the coprocessor.

The instruction can include as many as five effective address extension words following the first word in the cpRESTORE instruction format. These words contain any additional

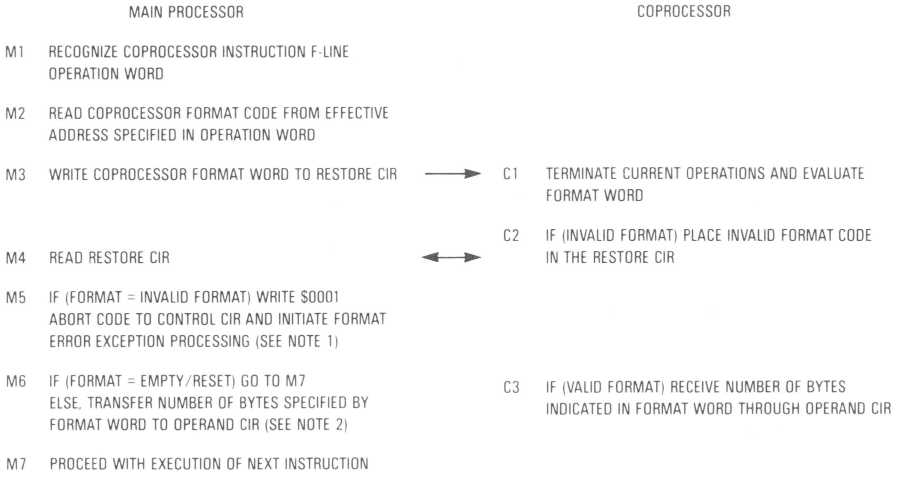
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Cp-ID			1	0	1	EFFECTIVE ADDRESS					
EFFECTIVE ADDRESS EXTENSION WORDS (0-5 WORDS)															

Figure 9-17. Coprocessor Context Restore Instruction Format (cpRESTORE)

information required to calculate the effective address specified by bits [5–0] of the operation word.

All memory addressing modes except the predecrement addressing mode are valid. Invalid effective address encodings cause the MC68020 to initiate F-line emulator exception processing (refer to **9.5.2.2 F-LINE EMULATOR EXCEPTIONS**).

9.2.3.4.2 Protocol. Figure 9-18 shows the protocol for the coprocessor context restore instruction. When the main processor executes a cpRESTORE instruction, it first reads the coprocessor format word from the effective address in the instruction. This format word contains a format code and a length field. During cpRESTORE operation, the main processor retains a copy of the length field to determine the number of bytes to be transferred to the coprocessor during the cpRESTORE operation and writes the format word to the restore CIR in order to initiate the coprocessor context restore.



MAIN PROCESSOR

M1 RECOGNIZE COPROCESSOR INSTRUCTION F-LINE OPERATION WORD

M2 READ COPROCESSOR FORMAT CODE FROM EFFECTIVE ADDRESS SPECIFIED IN OPERATION WORD

M3 WRITE COPROCESSOR FORMAT WORD TO RESTORE CIR

M4 READ RESTORE CIR

M5 IF (FORMAT = INVALID FORMAT) WRITE $0001 ABORT CODE TO CONTROL CIR AND INITIATE FORMAT ERROR EXCEPTION PROCESSING (SEE NOTE 1)

M6 IF (FORMAT = EMPTY/RESET) GO TO M7
ELSE, TRANSFER NUMBER OF BYTES SPECIFIED BY FORMAT WORD TO OPERAND CIR (SEE NOTE 2)

M7 PROCEED WITH EXECUTION OF NEXT INSTRUCTION

COPROCESSOR

C1 TERMINATE CURRENT OPERATIONS AND EVALUATE FORMAT WORD

C2 IF (INVALID FORMAT) PLACE INVALID FORMAT CODE IN THE RESTORE CIR

C3 IF (VALID FORMAT) RECEIVE NUMBER OF BYTES INDICATED IN FORMAT WORD THROUGH OPERAND CIR

NOTES: 1. See 9.6.1.5 FORMAT ERROR.
2. The MC68020 uses the length field in the format word read during M2 to determine the number of bytes to read from memory and write to the operand CIR.

Figure 9-18. Coprocessor Context Restore Instruction Protocol

When the coprocessor receives the format word in the restore CIR, it must terminate any current operations and evaluate the format word. If the format word represents a valid coprocessor context as determined by the coprocessor design, the coprocessor returns the format word to the main processor through the restore CIR and prepares to receive the number of bytes specified in the format word through its operand CIR.

After writing the format word to the restore CIR the main processor continues the cpRESTORE dialog by reading that same register. If the coprocessor returns a valid format word, the main processor transfers the number of bytes specified by the format word at the effective address to the operand CIR.

If the format word written to the restore CIR does not represent a valid coprocessor state frame, the coprocessor places an invalid format word in the restore CIR and terminates any current operations. The main processor receives the invalid format code, writes an abort mask (refer to **9.2.3.2.3 Invalid Format Word**) to the control CIR, and initiates format error exception processing (refer to **9.5.1.5 FORMAT ERRORS**).

The cpRESTORE instruction is a privileged instruction. When the main processor accesses a cpRESTORE instruction, it checks the supervisor bit in the status register. If the MC68020 attempts to execute a cpRESTORE instruction while at the user privilege level (status register bit [13] = 0), it initiates privilege violation exception processing without accessing any of the coprocessor interface registers (refer to **9.5.2.3 PRIVILEGE VIOLATIONS**).

9.3 COPROCESSOR INTERFACE REGISTER SET

The instructions of the M68000 coprocessor interface use registers of the CIR set to communicate with the coprocessor. These CIRs are not directly related to the coprocessor's programming model.

Figure 9-4 is a memory map of the CIR set. The registers denoted by asterisks (*) must be included in a coprocessor interface that implements coprocessor instructions in all four categories. The complete register model must be implemented if the system uses all of the coprocessor response primitives defined for the M68000 coprocessor interface.

The following paragraphs contain detailed descriptions of the registers.

9.3.1 Response CIR

The coprocessor uses the 16-bit response CIR to communicate all service requests (coprocessor response primitives) to the main processor. The main processor reads the response CIR to receive the coprocessor response primitives during the execution of instructions in the general and conditional instruction categories. The offset from the base address of the CIR set for the response CIR is $00. Refer to **9.4 COPROCESSOR RESPONSE PRIMITIVES**.

9.3.2 Control CIR

The main processor writes to the 2-bit control CIR to acknowledge coprocessor-requested exception processing or to abort the execution of a coprocessor instruction. The offset from the base address of the CIR set for the control CIR is $02. The control CIR occupies the two least significant bits of the word at that offset. The fourteen most significant bits of the word are undefined. Figure 9-19 shows the format of this register.

When the MC68020 receives one of the three take exception coprocessor response primitives, it acknowledges the primitive by writing the exception acknowledge mask (102) to the control CIR, which sets the exception acknowledge (XA) bit. The MC68020 writes

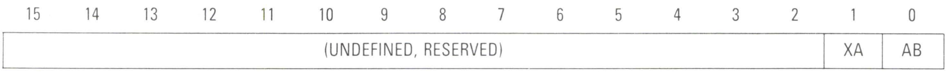


Figure 9-19. Control CIR Format

the abort mask (012), which sets the abort (AB) bit, to the control CIR to abort any coprocessor instruction that is in progress. (The most significant 14 bits of both masks are undefined.) The MC68020 aborts a coprocessor instruction when it detects one of the following exception conditions:

- An F-line emulator exception condition after reading a response primitive
- A privilege violation exception as it performs a supervisor check in response to a supervisor check primitive
- A format error exception when it receives an invalid format word or a valid format word that contains an invalid length

9.3.3 Save CIR

The coprocessor uses the 16-bit save CIR to communicate status and state frame format information to the main processor while executing a cpSAVE instruction. The main processor reads the save CIR to initiate execution of the cpSAVE instruction by the coprocessor. The offset from the base address of the CIR set for the save CIR is $04. Refer to **9.2.3.2 COPROCESSOR FORMAT WORDS**.

9.3.4 Restore CIR

The main processor initiates the cpRESTORE instruction by writing a coprocessor format word to the 16-bit restore register. During the execution of the cpRESTORE instruction, the coprocessor communicates status and state frame format information to the main processor through the restore CIR. The offset from the base address of the CIR set for the restore CIR is $06. Refer to **9.2.3.2 COPROCESSOR FORMAT WORDS**.

9.3.5 Operation Word CIR

The main processor writes the F-line operation word of the instruction in progress to the 16-bit operation word CIR in response to a transfer operation word coprocessor response primitive (refer to **9.4.6 Transfer Operation Word Primitive**).

The offset from the base address of the CIR set for the operation word CIR is $08.

9.3.6 Command CIR

The main processor initiates a general category instruction by writing the instruction command word, which follows the instruction F-line operation word in the instruction stream, to the 16-bit command CIR. The offset from the base address of the CIR set for the command CIR is $0A.

9.3.7 Condition CIR

The main processor initiates a conditional category instruction by writing the condition selector to the 16-bit condition CIR. The offset from the base address of the CIR set for the condition CIR is $0E. Figure 9-20 shows the format of the condition CIR.

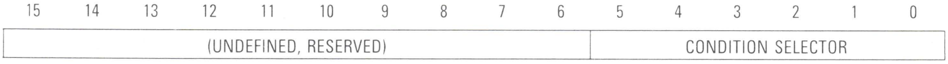

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
(UNDEFINED, RESERVED)
CONDITION SELECTOR

Figure 9-20. Condition CIR Format

9.3.8 Operand CIR

When the coprocessor requests the transfer of an operand, the main processor performs the transfer by reading from or writing to the 32-bit operand CIR. The offset from the base address of the CIR set for the operand CIR is $10.

The MC68020 aligns all operands transferred to and from the operand CIR to the most significant byte of this CIR. The processor performs a sequence of long-word transfers to read or write any operand larger than four bytes. If the operand size is not a multiple of four bytes, the portion remaining after the initial long-word transfers is aligned to the most significant byte of the operand CIR. Figure 9-21 shows the operand alignment used by the MC68020 when accessing the operand CIR.

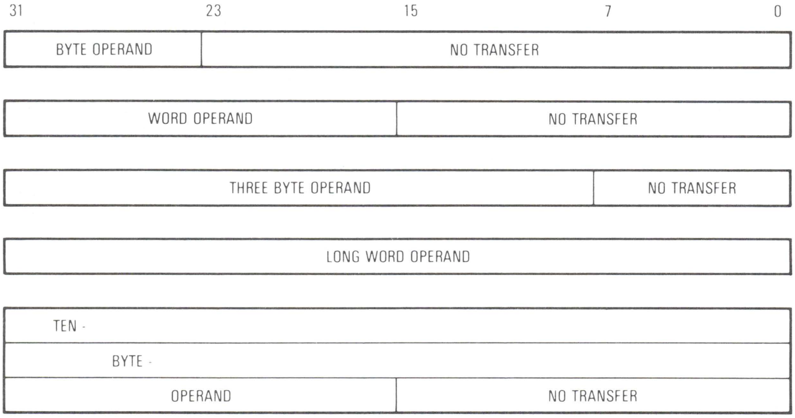

31
23
15
7
0
BYTE OPERAND
NO TRANSFER
WORD OPERAND
NO TRANSFER
THREE BYTE OPERAND
NO TRANSFER
LONG WORD OPERAND
TEN -
BYTE -
OPERAND
NO TRANSFER

Figure 9-21. Operand Alignment for Operand CIR Accesses

9.3.9 Register Select CIR

When the coprocessor requests the transfer of one or more main processor registers or a group of coprocessor registers, the main processor reads the 16-bit register select CIR to identify the number or type of registers to be transferred. The offset from the base address of the CIR set for the register select CIR is $14. The format of this register depends on the primitive that is currently using it. Refer to **9.4 COPROCESSOR RESPONSE PRIMITIVES**.

9.3.10 Instruction Address CIR

When the coprocessor requests the address of the instruction it is currently executing, the main processor transfers this address to the 32-bit instruction address CIR. Any transfer of the scanPC is also performed through the instruction address CIR (refer to **9.4.17 Transfer Status Register and ScanPC Primitive**). The offset from the base address of the CIR set for the instruction address CIR is $18.

9.3.11 Operand Address CIR

When a coprocessor requests an operand address transfer between the main processor and the coprocessor, the address is transferred through the 32-bit operand address CIR. The offset from the base address of the CIR set for the operand address CIR is $1C.

9.4 COPROCESSOR RESPONSE PRIMITIVES

The response primitives are primitive instructions that the coprocessor issues to the main processor during the execution of a coprocessor instruction. The coprocessor uses response primitives to communicate status information and service requests to the main processor. In response to an instruction command word written to the command CIR or a condition selector in the condition CIR, the coprocessor returns a response primitive in the response CIR. Within the general and conditional instruction categories, individual instructions are distinguished by the operation of the coprocessor hardware and, also by services specified by coprocessor response primitives and provided by the main processor.

Subsequent paragraphs, beginning with **9.4.2 Coprocessor Response Primitive General Format**, consist of detailed descriptions of the M68000 coprocessor response primitives supported by the MC68020. Any response primitive that the MC68020 does not recognize causes it to initiate protocol violation exception processing (refer to **9.5.2.1 PROTOCOL VIOLATIONS**). This processing of undefined primitives supports emulation of extensions to the M68000 coprocessor response primitive set by the protocol violation exception handler. Exception processing related to the coprocessor interface is discussed in **9.5 EXCEPTIONS**.

9.4.1 ScanPC

Several of the response primitives involve the scanPC, and many of them require the main processor to use it while performing services requested. These paragraphs describe the scanPC and tell how it operates.

During the execution of a coprocessor instruction, the program counter in the MC68020 contains the address of the F-line operation word of that instruction. A second register, called the scanPC, sequentially addresses the remaining words of the instruction.

If the main processor requires extension words in order to calculate an effective address or destination address of a branch operation, it uses the scanPC to address these extension words in the instruction stream. Also, if a coprocessor requests the transfer of extension words, the scanPC addresses the extension words during the transfer. As the processor references each word, it increments the scanPC to point to the next word in the instruction stream. When an instruction is completed, the processor transfers the value in the scanPC to the program counter to address the operation word of the next instruction.

The value in the scanPC when the main processor reads the first response primitive after beginning to execute an instruction depends on the instruction being executed. For a cpGEN instruction, the scanPC points to the word following the coprocessor command word. For the cpBcc instructions, the scanPC points to the word following the instruction F-line operation word. For the cpScc, cpTRAPcc, and cpDBcc instructions, the scanPC points to the word following the coprocessor condition specifier word.

If a coprocessor implementation uses optional instruction extension words with a general or conditional instruction, the coprocessor must use these words consistently so that the scanPC is updated accordingly during the instruction execution. Specifically, during the execution of general category instructions, when the coprocessor terminates the instruction protocol, the MC68020 assumes that the scanPC is pointing to the operation word of the next instruction to be executed. During the execution of conditional category instructions, when the coprocessor terminates the instruction protocol, the MC68020 assumes that the scanPC is pointing to the word following the last of any coprocessor defined extension words in the instruction format.

9.4.2 Coprocessor Response Primitive General Format

The M68000 coprocessor response primitives are encoded in a 16-bit word that is transferred to the main processor through the response CIR. Figure 9-22 shows the format of the coprocessor response primitives.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	FUNCTION					PARAMETER							

Figure 9-22. Coprocessor Response Primitive Format

The encoding of bits [12–0] of a coprocessor response primitive depends on the individual primitive. Bits [15–13], however, specify optional additional operations that apply to most of the primitives defined for the M68000 coprocessor interface.

Bit [15], the CA bit, specifies the come-again operation of the main processor. When the main processor reads a response primitive from the response CIR with the come-again bit set to one, it performs the service indicated by the primitive and then reads the response CIR again. Using the CA bit, a coprocessor can transfer several response primitives to the main processor during the execution of a single coprocessor instruction.

Bit [14], the PC bit, specifies the pass program counter operation. When the main processor reads a primitive with the PC bit set from the response CIR, the main processor immediately passes the current value in its program counter to the instruction address CIR as the first operation in servicing the primitive request. The value in the program counter is the address of the F-line operation word of the coprocessor instruction currently executing. The PC bit is implemented in all of the coprocessor response primitives currently defined for the M68000 coprocessor interface.

When an undefined primitive or a primitive that requests an illegal operation is passed to the main processor, the main processor initiates exception processing for either an F-line emulator or a protocol violation exception (refer to **9.5.2 Main Processor Detected Exceptions**). If the PC bit is set in one of these response primitives, however, the main processor passes the program counter to the instruction address CIR before it initiates exception processing.

When the main processor initiates a cpGEN instruction that can be executed concurrently with main processor instructions, the PC bit is usually set in the first primitive returned by the coprocessor. Since the main processor proceeds with instruction stream execution once the coprocessor releases it, the coprocessor must record the instruction address to support any possible exception processing related to the instruction. Exception processing related to concurrent coprocessor instruction execution is discussed in **9.5.1 Coprocesssor Detected Exceptions**.

Bit [13], the DR bit, is the direction bit. It applies to operand transfers between the main processor and the coprocessor. If DR = 0, the direction of transfer is from the main processor to the coprocessor (main processor write). If DR = 1, the direction of transfer is from the coprocessor to the main processor (main processor read). If the operation indicated by a given response primitive does not involve an explicit operand transfer, the value of this bit depends on the particular primitive encoding.

9.4.3 Busy Primitive

The busy response primitive causes the main processor to reinitiate a coprocessor instruction. This primitive applies to instructions in the general and conditional categories. Figure 9-23 shows the format of the busy primitive.

This primitive uses the PC bit as previously described.

Coprocessors that can operate concurrently with the main processor but cannot buffer write operations to their command or condition CIR use the busy primitive. A coprocessor

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	PC	1	0	0	1	0	0	0	0	0	0	0	0	0	0

Figure 9-23. Busy Primitive Format

may execute a cpGEN instruction concurrently with an instruction in the main processor. If the main processor attempts to initiate an instruction in the general or conditional instruction category while the coprocessor is concurrently executing a cpGEN instruction, the coprocessor can place the busy primitive in the response CIR. When the main processor reads this primitive, it services pending interrupts (using a preinstruction exception stack frame, refer to Figure 10-41). The processor then restarts the general or conditional coprocessor instruction that it had attempted to initiate earlier.

The busy primitive should only be used in response to a write to the command or condition CIR. It should be the first primitive returned after the main processor attempts to initiate a general or conditional category instruction. In particular, the busy primitive should not be issued after program-visible resources have been altered by the instruction. (Program visible resources include coprocessor and main processor program-visible registers and operands in memory, but not the scanPC.) The restart of an instruction after it has altered program-visible resources causes those resources to have inconsistent values when the processor reinitiates the instruction.

The MC68020 responds to the busy primitive differently in a special case that can occur during a breakpoint operation (refer to **8.1.12 Multiple Exceptions**). This special case occurs when a breakpoint acknowledge cycle initiates a coprocessor F-line instruction, the coprocessor returns the busy primitive in response to the instruction initiation, and an interrupt is pending. When these three conditions are met, the processor re-executes the breakpoint acknowledge cycle after the interrupt exception processing has been completed. A design that uses a breakpoint to monitor the number of passes through a loop by incrementing or decrementing a counter may not work correctly under these conditions. This special case may cause several breakpoint acknowledge cycles to be executed during a single pass through a loop.

9.4.4 Null Primitive

The null coprocessor response primitive communicates coprocessor status information to the main processor. This primitive applies to instructions in the general and conditional categories. Figure 9-24 shows the format of the null primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	1	0	0	IA	0	0	0	0	0	0	PF	TF

Figure 9-24. Null Primitive Format

This primitive uses the CA and PC bits as previously described.

Bit [8], the IA bit, specifies the interrupts allowed optional operation. This bit determines whether the MC68020 services pending interrupts prior to rereading the response CIR after receiving a null primitive. Interrupts are allowed when the IA bit is set.

Bit [1], the PF bit, shows the processing-finished status of the coprocessor. That is, PF = 1 indicates that the coprocessor has completed all processing associated with an instruction.

Bit [0], the TF bit, indicates the true/false condition during the execution of a conditional category instruction. TF = 1 is the true condition specifier and TF = 0 is the false condition specifier. The TF bit is only relevant for null primitives with CA = 0 that are used by the coprocessor during the execution of a conditional instruction.

The MC68020 processes a null primitive with CA = 1 in the same manner whether executing a general or conditional category coprocessor instruction. If the coprocessor sets CA and IA to one in the null primitive, the main processor services pending interrupts (using a mid-instruction stack frame, refer to Figure 9-43) and reads the response CIR again. If the coprocessor sets CA to one and IA to zero in the null primitive, the main processor reads the response CIR again without servicing any pending interrupts.

A null, CA = 0 primitive provides a condition evaluation indicator to the main processor during the execution of a conditional instruction and ends the dialogue between the main processor and coprocessor for that instruction. The main processor completes the execution of a conditional category coprocessor instruction when it receives the primitive. The PF bit is not relevant during conditional instruction execution since the primitive itself implies completion of processing.

Usually, when the main processor reads any primitive that does not have CA = 1 while executing a general category instruction, it terminates the dialogue between the main processor and coprocessor. If a trace exception is pending, however, the main processor does not terminate the instruction dialogue until it reads a null, CA = 0, PF = 1 primitive from the response CIR (refer to **9.5.2.5 TRACE EXCEPTIONS**). Thus, the main processor continues to read the response CIR until it receives a null, CA = 0, PF = 1 primitive, and then performs trace exception processing. When IA = 1, the main processor services pending interrupts before reading the response CIR again.

A coprocessor can be designed to execute a cpGEN instruction concurrently with the execution of main processor instructions and, also, buffer one write operation to either its command or condition CIR. This type of coprocessor issues a null primitive with CA = 1 when it is concurrently executing a cpGEN instruction, and the main processor initiates another general or conditional coprocessor instruction. This primitive indicates that the coprocessor is busy and the main processor should read the response CIR again without reinitiating the instruction. The IA bit of this null primitive usually should be set in order to minimize interrupt latency while the main processor is waiting for the coprocessor to complete the general category instruction.

Table 9-3 summarizes the encodings of the null primitive.

9.4.5 Supervisor Check Primitive

The supervisor check primitive verifies that the main processor is operating in the supervisor state while executing a coprocessor instruction. This primitive applies to instructions in the general and conditional coprocessor instruction categories. Figure 9-25 shows the format of the supervisor check primitive.

This primitive uses the PC bit as previously described. Bit [15] is shown as 1, but during execution of a general category instruction this primitive performs the same operations

Table 9-3. Null Coprocessor Response Primitive Encodings

CA	PC	IA	PF	TF	General Instructions	Conditional Instructions
x	1	x	x	x	Pass Program Counter to Instruction Address CIR, Clear PC Bit and Proceed with Operation Specified by CA, IA, PF, and TF Bits	Same as general Category
1	0	0	x	x	Reread Response CIR, Do Not Service Pending Interrupts	Same as General Category
1	0	1	x	x	Service Pending Interrupts and Reread the Response CIR	Same as General Category
0	0	0	0	c	If (Trace Pending) Reread Response CIR Else, Execute Next Instruction	Main Processor Completes Instruction Execution Based on TF = c
0	0	1	0	c	If (Trace Pending) Service Pending Interrupts and Reread Response CIR Else, Execute Next Instruction	Main Processor Completes Instruction Execution Based on TF = c
0	0	x	1	c	Coprocessor Instruction Completed; Service Pending Exceptions or Execute Next Instruction	Main Processor Completes Instruction Execution Based on TF = c.

x = Don't Care
c = 1 or 0 Depending on Coprocessor Condition Evaluation

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	PC	0	0	0	1	0	0	0	0	0	0	0	0	0	0

Figure 9-25. Supervisor Check Primitive Format

regardless of the value of bit [15]. If this primitive is issued with bit [15] = 0 during a conditional category instruction, however, the main processor initiates protocol violation exception processing.

When the main processor reads the supervisor check primitive from the response CIR, it checks the value of the S bit in the status register. If S = 0 (main processor operating at user privilege level), the main processor aborts the coprocessor instruction by writing an abort mask (refer to **9.3.2 Control CIR**) to the control CIR. The main processor then initiates privilege violation exception processing (refer to **9.5.2.3 PRIVILEGE VIOLATIONS**). If the main processor is at the supervisor privilege level when it receives this primitive, it reads the response CIR again.

The supervisor check primitive allows privileged instructions to be defined in the coprocessor general and conditional instruction categories. This primitive should be the first one issued by the coprocessor during the dialog for an instruction that is implemented as privileged.

9.4.6 Transfer Operation Word Primitive

The transfer operation word primitive requests a copy of the coprocessor instruction operation word for the coprocessor. This primitive applies to general and conditional category instructions. Figure 9-26 shows the format of the transfer operation word primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	0	1	1	1	0	0	0	0	0	0	0	0

Figure 9-26. Transfer Operation Word Primitive Format

This primitive uses the CA and PC bits as previously described. If this primitive is issued with CA = 0 during a conditional category instruction, the main processor initiates protocol violation exception processing.

When the main processor reads this primitive from the response CIR, it transfers the F-line operation word of the currently executing coprocessor instruction to the operation word CIR. The value of the scanPC is not affected by this primitive.

9.4.7 Transfer from Instruction Stream Primitive

The transfer from instruction stream primitive initiates transfers of operands from the instruction stream to the coprocessor. This primitive applies to general and conditional category instructions. Figure 9-27 shows the format of the transfer from instruction stream primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	1	1	1	1	LENGTH							

Figure 9-27. Transfer From Instruction Stream Primitive Format

This primitive uses the CA and PC bits as previously described. If this primitive is issued with CA = 0 during a conditional category instruction, the main processor initiates protocol violation exception processing.

Bits [7–0] of the primitive format specify the length, in bytes, of the operand to be transferred from the instruction stream to the coprocessor. The length must be an even number of bytes. If an odd length is specified, the main processor initiates protocol violation exception processing (refer to **9.5.2.1 PROTOCOL VIOLATIONS**).

This primitive transfers coprocessor defined extension words to the coprocessor. When the main processor reads this primitive from the response CIR, it copies the number of bytes indicated by the length field from the instruction stream to the operand CIR. The first word or long word transferred is at the location pointed to by the scanPC when the primitive is read by the main processor, and the scanPC is incremented after each word or long word is transferred. When execution of the primitive has completed, the scanPC has been incremented by the total number of bytes transferred and points to the word following the last word transferred. The main processor transfers the operands from the instruction stream using a sequence of long-word writes to the operand CIR. If the length field is not an even multiple of four bytes, the last two bytes from the instruction stream are transferred using a word write to the operand CIR.

9.4.8 Evaluate and Transfer Effective Address Primitive

The evaluate and transfer effective address primitive evaluates the effective address specified in the coprocessor instruction operation word and transfers the result to the coprocessor. This primitive applies to general category instructions. If this primitive is issued by the coprocessor during the execution of a conditional category instruction, the main processor initiates protocol violation exception processing. Figure 9-28 shows the format of the evaluate and transfer effective address primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	1	0	1	0	0	0	0	0	0	0	0	0

Figure 9-28. Evaluate and Transfer Effective Address Primitive Format

This primitive uses the CA and PC bits as previously described.

When the main processor reads this primitive while executing a general category instruction, it evaluates the effective address specified in the instruction. At this point, the scanPC contains the address of the first of any required effective address extension words. The main processor increments the scanPC by two after it references each of these extension words. After the effective address is calculated, the resulting 32-bit value is written to the operand address CIR.

The MC68020 only calculates effective addresses for control-alterable addressing modes in response to this primitive. If the addressing mode in the operation word is not a control-alterable mode, the main processor aborts the instruction by writing a $0001 to the control CIR and initiates F-line emulation exception processing (refer to **9.5.2.2 F-LINE EMULATOR EXCEPTIONS**).

9.4.9 Evaluate Effective Address and Transfer Data Primitive

The evaluate effective address and transfer data primitive transfers an operand between the coprocessor and the effective address specified in the coprocessor instruction operation word. This primitive applies to general category instructions. If the coprocessor issues this primitive during the execution of a conditional category instruction, the main processor initiates protocol violation exception processing. Figure 9-29 shows the format of the evaluate effective address and transfer data primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	1	0	VALID EA			LENGTH							

Figure 9-29. Evaluate Effective Address and Transfer Data Primitive Format

This primitive uses the CA, PC, and DR bits as previously described .

The valid effective address field (bits [10–8]) of the primitive format specifies the valid effective address categories for this primitive. If the effective address specified in the instruction operation word is not a member of the class specified by bits [10–8], the main processor aborts the coprocessor instruction by writing an abort mask (refer to **9.3.2 Control CIR**) to the control CIR, and by initiating F-line emulation exception processing. Table 9-4 lists the valid effective address field encodings.

Table 9-4. Valid Effective Address Codes

000	Control Alterable
001	Data Alterable
010	Memory Alterable
011	Alterable
100	Control
101	Data
110	Memory
111	Any Effective Address (No Restriction)

Even when the valid effective address fields specified in the primitive and in the instruction operation word match, the MC68020 initiates protocol violation exception processing if the primitive requests a write to a nonalterable effective address.

The length in bytes of the operand to be transferred is specified by bits [7–0] of the primitive format. Several restrictions apply to operand lengths for certain effective addressing modes. If the effective address is a main processor register (register direct mode), only operand lengths of one, two, or four bytes are valid; all other lengths (zero, for example) cause the main processor to initiate protocol violation exception processing. Operand lengths of zero through 255 bytes are valid for the memory addressing modes.

The length of zero through 255 bytes does not apply to an immediate operand. The length of an immediate operand must be one byte or an even number of bytes (less than 256), and the direction of transfer must be to the coprocessor; otherwise, the main processor initiates protocol violation exception processing.

When the main processor receives this primitive during the execution of a general category instruction, it verifies that the effective address encoded in the instruction operation word is in the category specified by the primitive. If so, the processor calculates the effective address using the appropriate effective address extension words at the current scanPC address and increments the scanPC by two for each word referenced. The main processor then transfers the number of bytes specified in the primitive between the operand CIR and the effective address using long-word transfers whenever possible. Refer to **9.3.8 Operand CIR** for information concerning operand alignment for transfers involving the operand CIR.

The DR bit specifies the direction of the operand transfer. DR = 0 requests a transfer from the effective address to the operand CIR and DR = 1 specifies a transfer from the operand CIR to the effective address.

If the effective addressing mode specifies the predecrement mode, the address register used is decremented by the size of the operand before the transfer. The bytes within the operand are then transferred to or from ascending addresses beginning with the location specified by the decremented address register. In this mode, if A7 is used as the address register and the operand length is one byte, A7 is decremented by two to maintain a word aligned stack.

For the postincrement effective addressing mode, the address register used is incremented by the size of the operand after the transfer. The bytes within the operand are transferred to or from ascending addresses beginning with the location specified by the address register. In this mode, if A7 is used as the address register and the operand length is one byte, A7 is incremented by two after the transfer to maintain a word aligned stack. Transferring odd length operands longer than one byte using the –(A7) or (A7)+ addressing modes can result in a stack pointer that is not word aligned.

The processor repeats the effective address calculation each time this primitive is issued during the execution of a given instruction. The calculation uses the current contents of any required address and data registers. The instruction must include a set of effective address extension words for each repetition of a calculation that requires them. The processor locates these words at the current scanPC location and increments the scanPC by two for each word referenced in the instruction stream.

The MC68020 sign extends a byte or word size operand to a long-word value when it is transferred to an address register (A7–A0) using this primitive with the register direct effective addressing mode. A byte or word-size operand transferred to a data register (D7–D0) only overwrites the lower byte or word of the data register.

9.4.10 Write to Previously Evaluated Effective Address Primitive

The write to previously evaluated effective address primitive transfers an operand from the coprocessor to a previously evaluated effective address. This primitive applies to general category instructions. If the coprocessor uses this primitive during the execution of a conditional category instruction, the main processor initiates protocol violation exception processing. Figure 9-30 shows the format of the write to previously evaluated effective address primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	1	0	0	0	0	0	LENGTH							

Figure 9-30. Write to Previously Evaluated Effective Address Primitive Format

This primitive uses the CA and PC bits as previously described.

Bits [7–0] of the primitive format specify the length of the operand in bytes. The MC68020 transfers operands between zero and 255 bytes in length.

When the main processor receives this primitive during the execution of a general category instruction, it transfers an operand from the operand CIR to an effective address specified by a temporary register within the MC68020. When a previous primitive for the current instruction has evaluated the effective address, this temporary register contains the evaluated effective address. Primitives that store an evaluated effective address in a temporary register of the main processor are the evaluate and transfer effective address, evaluate effective address and transfer data, and transfer multiple coprocessor registers primitives. If this primitive is used during an instruction in which the effective address specified in the instruction operation word has not been calculated, the effective address used for the write is undefined. Also, if the previously evaluated effective address was register direct, the address written to in response to this primitive is undefined.

The function code value during the write operation indicates either supervisor or user data space depending on the value of the S bit in the MC68020 status register when the processor reads this primitive. While a coprocessor should request writes to only alterable effective addressing modes, the MC68020 does not check the type of effective address used with this primitive. For example, if the previously evaluated effective address was program counter relative and the MC68020 is at the user privilege level ($S=0$ in status register), the MC68020 writes to user data space at the previously calculated program relative address (the 32-bit value in the temporary internal register of the processor).

Operands longer than four bytes are transferred in increments of four bytes (operand parts) when possible. The main processor reads a long-word operand part from the operand CIR and transfers this part to the current effective address. The transfers continue in this manner using ascending memory locations until all of the long-word operand parts are transferred, and any remaining operand part is then transferred using a one-, two-, or three-byte transfer as required. The operand parts are stored in memory using ascending addresses beginning with the address in the MC68020 temporary register.

The execution of this primitive does not modify any of the registers in the MC68020 programmer's model, even if the previously evaluated effective address mode is the predecrement or postincrement mode. If the previously evaluated effective addressing mode used any of the MC68020 internal address or data registers, the effective address value used is the final value from the preceding primitive. That is, this primitive uses the value from an evaluate and transfer effective address, evaluate effective address and transfer data, or transfer multiple coprocessor registers primitive without modification.

The take address and transfer data primitive described in the next section does not replace the effective address value that has been calculated by the MC68020. The address that the main processor obtains in response to the take address and transfer data primitive is not available to the write to previously evaluated effective address primitive.

A coprocessor can issue an evaluate effective address and transfer data primitive followed by this primitive to perform a read-modify-write operation that is not indivisible. The bus cycles for this operation are normal bus cycles that can be interrupted, and the bus can be arbitrated between the cycles.

9.4.11 Take Address and Transfer Data Primitive

The take address and transfer data primitive transfers an operand between the coprocessor and an address supplied by the coprocessor. This primitive applies to general and conditional category instructions. Figure 9-31 shows the format of the take address and transfer data primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	1	0	1	LENGTH							

Figure 9-31. Take Address and Transfer Data Primitive Format

This primitive uses the CA, PC, and DR bits as previously described. If the coprocessor issues this primitive with CA = 0 during a conditional category instruction, the main processor initiates protocol violation exception processing.

Bits [7–0] of the primitive format specify the operand length, which can be from zero to 255 bytes.

The main processor reads a 32-bit address from the operand address CIR. The processor transfers the operand, using a series of long-word transfers, between this address and the operand CIR. The DR bit determines the direction of the transfer. The processor reads or writes the operand parts to ascending addresses starting at the address from the operand address CIR. If the operand length is not a multiple of four bytes, the final operand part is transferred using a one-, two-, or three-byte transfer as required.

The function code used with the address read from the operand address CIR indicates either supervisor or user data space according to the value of the S bit in the MC68020 status register.

9.4.12 Transfer to/from Top of Stack Primitive

The transfer to/from top-of-stack primitive transfers an operand between the coprocessor and the top of the currently active main processor stack (refer to **2.8.1 System Stack**). This primitive applies to general and conditional category instructions. Figure 9-32 shows the format of the transfer to/from top of stack primitive.

This primitive uses the CA, PC, and DR bits as previously described. If the coprocessor issues this primitive with CA = 0 during a conditional category instruction, the main processor initiates protocol violation exception processing.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	1	1	1	0	LENGTH							

Figure 9-32. Transfer to/from Top of Stack Primitive Format

Bits [7–0] of the primitive format specify the length in bytes of the operand to be transferred. The operand may be one, two, or four bytes in length; other length values cause the main processor to initiate protocol violation exception processing.

If DR = 0, the main processor transfers the operand from the currently active system stack to the operand CIR. The implied effective address mode used for the transfer is the (A7) + addressing mode. A one-byte operand causes the stack pointer to be incremented by two after the transfer to maintain word alignment of the stack.

If DR = 1, the main processor transfers the operand from the operand CIR to the currently active stack. The implied effective address mode used for the transfer is the – (A7) addressing mode. A one-byte operand causes the stack pointer to be decremented by two before the transfer to maintain word alignment of the stack.

9.4.13 Transfer Single Main Processor Register Primitive

The transfer single main processor register primitive transfers an operand between one of the main processor's data or address registers and the coprocessor. This primitive applies to general and conditional category instructions. Figure 9-33 shows the format of the transfer single main processor register primitive.

This primitive uses the CA, PC, and DR bits as previously described. If the coprocessor issues this primitive with CA = 0 during a conditional category instruction, the main processor initiates protocol violation exception processing.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	1	1	0	0	0	0	0	0	D/A	REGISTER		

Figure 9-33. Transfer Single Main Processor Register Primitive Format

Bit [3], the D/A bit, specifies whether the primitive transfers an address or data register. D/A = 0 indicates a data register and D/A = 1 indicates an address register. Bits [2-0] contain the register number.

If DR = 0, the main processor writes the long-word operand in the specified register to the operand CIR. If DR = 1, the main processor reads a long-word operand from the operand CIR and transfers it to the specified data or address register.

9.4.14 Transfer Main Processor Control Register Primitive

The transfer main processor control register primitive transfers a long-word operand between one of its control registers and the coprocessor. This primitive applies to general and conditional category instructions. Figure 9-34 shows the format of the transfer main processor control register primitive. This primitive uses the CA, PC, and DR bits as previously described. If the coprocessor issues this primitive with CA = 0 during a conditional category instruction, the main processor initiates protocol violation exception processing.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	1	1	0	1	0	0	0	0	0	0	0	0

Figure 9-34. Transfer Main Processor Control Register Primitive Format

When the main processor receives this primitive, it reads a control register select code from the register select CIR. This code determines which main processor control register is transferred. Table 9-5 lists the valid control register select codes. If the control register select code is not valid, the MC68020 initiates protocol violation exception processing (refer to **9.5.2.1 PROTOCOL VIOLATIONS**).

Table 9-5. Main Processor Control Register Selector Codes

Hex	Control Register
x000	Source Function Code (SFC) Register
x001	Destination Function Code (DFC) Register
x002	Cache Control Register (CACR)
x800	User Stack Pointer (USP)
x801	Vector Base Register (VBR)
x802	Cache Address Register (CAAR)
x803	Master Stack Pointer (MSP)
x804	Interrupt Stack Pointer (ISP)
All other codes cause a protocol violation exception	

After reading a valid code from the register select CIR, if DR = 0, the main processor writes the long-word operand from the specified control register to the operand CIR. If DR = 1, the main processor reads a long-word operand from the operand CIR and places it in the specified control register.

9.4.15 Transfer Multiple Main Processor Registers Primitive

The transfer multiple main processor registers primitive transfers long-word operands between one or more of its data or address registers and the coprocessor. This primitive applies to general and conditional category instructions. Figure 9-35 shows the format of the transfer multiple main processor registers primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	1	1	0	0	0	0	0	0	0	0	0

Figure 9-35. Transfer Multiple Main Processor Registers Primitive Format

This primitive uses the CA, PC, and DR bits as previously described. If the coprocessor issues this primitive with CA=0 during a conditional category instruction, the main processor initiates protocol violation exception processing.

When the main processor receives this primitive it reads a 16-bit register select mask from the register select CIR. The format of the register select mask is shown in Figure 9-36. A register is transferred if the bit corresponding to the register in the register select mask is set to one. The selected registers are transferred in the order D7–D0 and then A7–A0.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

Figure 9-36. Register Select Mask Format

If DR=0, the main processor writes the contents of each register indicated in the register select mask to the operand CIR using a sequence of long-word transfers. If DR=1, the main processor reads a long-word operand from the operand CIR into each register indicated in the register select mask. The registers are transferred in the same order regardless of the direction of transfer indicated by the DR bit.

9.4.16 Transfer Multiple Coprocessor Registers Primitive

The transfer multiple coprocessor registers primitive transfers from zero to sixteen operands between the effective address specified in the coprocessor instruction and the coprocessor. This primitive applies to general category instructions. If the coprocessor issues this primitive during the execution of a conditional category instruction, the main processor initiates protocol violation exception processing. Figure 9-37 shows the format of the transfer multiple coprocessor registers primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	0	0	1	LENGTH							

Figure 9-37. Transfer Multiple Coprocessor Registers Primitive Format

This primitive uses the CA, PC, and DR bits as previously described.

Bits [7–0] of the primitive format indicate the length in bytes of each operand transferred. The operand length must be an even number of bytes; odd-length operands cause the MC68020 to initiate protocol violation exception processing (refer to **9.5.2.1 PROTOCOL VIOLATIONS**).

When the main processor reads this primitive, it calculates the effective address specified in the coprocessor instruction. The scanPC should be pointing to the first of any necessary effective address extension words when this primitive is read from the response CIR; the scanPC is incremented by two for each extension word referenced during the effective address calculation. For transfers from the effective address to the coprocessor (DR = 0), the control addressing modes and the postincrement addressing mode are valid. For transfers from the coprocessor to the effective address (DR = 1), the control alterable and predecrement addressing modes are valid. Invalid addressing modes cause the MC68020 to abort the instruction by writing an abort mask (refer to **9.3.2 Control CIR**) to the control CIR and to initiate F-line emulator exception processing (refer to **9.5.2.2 F-LINE EMULATOR EXCEPTIONS**).

After performing the effective address calculation, the MC68020 reads a 16-bit register select mask from the register select CIR. The coprocessor uses the register select mask to specify the number of operands to transfer; the MC68020 counts the number of ones in the register select mask to determine the number of operands. The order of the ones in the register select mask is not relevant to the operation of the main processor. As many as 16 operands can be transferred by the main processor in response to this primitive. The total number of bytes transferred is the product of the number of operands transferred and the length of each operand specified in bits [7–0] of the primitive format.

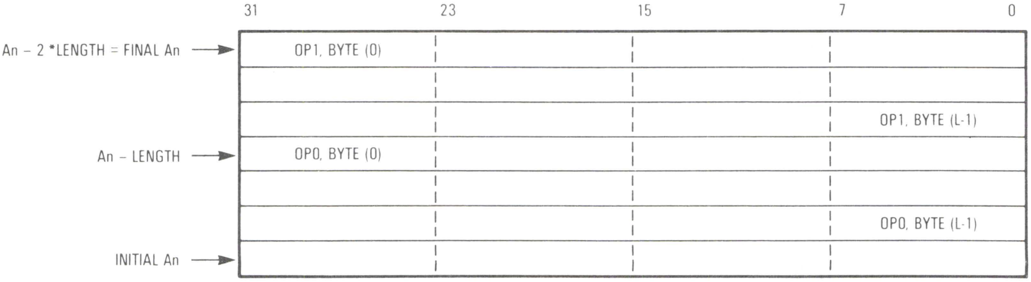
If DR = 1, the main processor reads the number of operands specified in the register select mask from the operand CIR and writes these operands to the effective address specified in the instruction using long-word transfers whenever possible. If DR = 0, the main processor reads the number of operands specified in the register select mask from the effective address and writes them to the operand CIR.

For the control addressing modes, the operands are transferred to or from memory using ascending addresses. For the postincrement addressing mode, the operands are read from memory with ascending addresses also, and the address register used is incremented by the size of an operand after each operand is transferred. The address register used with the (An)+ addressing mode is incremented by the total number of bytes transferred during the primitive execution.

For the predecrement addressing mode, the operands are written to memory with descending addresses, but the bytes within each operand are written to memory with ascending addresses. As an example, Figure 9-38 shows the format in long-word oriented memory for two 12-byte operands transferred from the coprocessor to the effective address using the −(An) addressing mode. The processor decrements the address register by the size of an operand before the operand is transferred. It writes the bytes of the operand to ascending memory addresses. When the transfer is complete, the address register has been decremented by the total number of bytes transferred. The MC68020 transfers the data using long-word transfers whenever possible.

9.4.17 Transfer Status Register and ScanPC Primitive

Both the transfer status register and the scanPC primitive transfers values between the coprocessor and the main processor status register. On an optional basis, the scanPC also makes transfers. This primitive applies to general category instructions. If the coprocessor issues this primitive during the execution of a conditional category instruction, the main processor initiates protocol violation exception processing. Figure 9-39 shows the format of the transfer status register and scanPC primitive.



NOTE: OP0, Byte (0) is the first byte written to memory
OP0, Byte (L-1) is the last byte of the first operand written to memory
OP1, Byte (0) is the first byte of the second operand written to memory
OP1, Byte (L-1) is the last byte written to memory

Figure 9-38. Operand Format in Memory for Transfer to –(An)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	0	1	SP	0	0	0	0	0	0	0	0

Figure 9-39. Transfer Status Register and ScanPC Primitive Format

This primitive uses the CA, PC, and DR bits as previously described.

Bit [8], the SP bit, selects the scanPC option. If SP = 1, the primitive transfers both the scanPC and status register. If SP = 0, only the status register is transferred.

If SP = 0 and DR = 0, the main processor writes the 16-bit status register value to the operand CIR. If SP = 0 and DR = 1, the main processor reads a 16-bit value from the operand CIR into the main processor status register.

If SP = 1 and DR = 0, the main processor writes the long-word value in the scanPC to the instruction address CIR and then writes the status register value to the operand CIR. If SP = 1 and DR = 1, the main processor reads a 16-bit value from the operand CIR into the status register and then reads a long-word value from the instruction address CIR into the scanPC.

With this primitive, a general category instruction can change the main processor program flow by placing a new value in the status register, in the scanPC, or new values in both the status register and the scanPC. By accessing the status register, the coprocessor can determine and manipulate the main processor condition codes, supervisor status, trace modes, selection of the active stack, and interrupt mask level.

The MC68020 discards any instruction words that have been prefetched beyond the current scanPC location when this primitive is issued with DR = 1 (transfer to main processor). The

MC68020 then refills the instruction pipe from the scanPC address in the address space indicated by the status register S bit.

If the MC68020 is operating in the trace on change of flow mode (T1:T0 in the status register contains 01) when the coprocessor instruction begins to execute, and this primitive is issued with DR = 1 (from coprocessor to main), the MC68020 prepares to take a trace exception. The trace exception occurs when the coprocessor signals that it has completed all processing associated with the instruction. Changes in the trace modes due to the transfer of the status register to main take effect on execution of the next instruction.

9.4.18 Take Preinstruction Exception Primitive

The take preinstruction exception primitive initiates exception processing using a coprocessor-supplied exception vector number and the preinstruction exception stack frame format. This primitive applies to general and conditional category instructions. Figure 9-40 shows the format of the take preinstruction exception primitive.

15	14	13	12	11	10	9	8	7–0
0	PC	0	1	1	1	0	0	VECTOR NUMBER

Figure 9-40. Take Preinstruction Exception Primitive Format

The primitive uses the PC bit as previously described. Bits [7–0] contain the exception vector number used by the main processor to initiate exception processing.

When the main processor receives this primitive, it acknowledges the coprocessor exception request by writing an exception acknowledge mask (refer to **9.3.2 Control CIR**) to the control CIR. The MC68020 then proceeds with exception processing as described in **8.1 EXCEPTION PROCESSING SEQUENCE**. The vector number for the exception is taken from bits [7–0] of the primitive, and the MC68020 uses the four-word stack frame format shown in Figure 9-41.

The value of the program counter saved in this stack frame is the F-line operation word address of the coprocessor instruction during which the primitive was received. Thus, if the exception handler routine does not modify the stack frame, an RTE instruction causes the MC68020 to return and reinitiate execution of the coprocessor instruction.

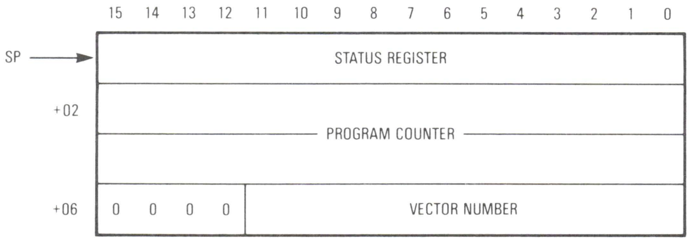


Figure 9-41. MC68020 Preinstruction Stack Frame

The take preinstruction exception primitive can be used when the coprocessor does not recognize a value written to either its command CIR or condition CIR to initiate a coprocessor instruction. This primitive can also be used if an exception occurs in the coprocessor instruction before any program-visible resources are modified by the instruction operation. This primitive should not be used during a coprocessor instruction if program-visible resources have been modified by that instruction. Otherwise, since the MC68020 reinitiates the instruction when it returns from exception processing, the restarted instruction receives the previously modified resources in an inconsistent state.

One of the most important uses of the take preinstruction exception primitive is to signal an exception condition in a cpGEN instruction that was executing concurrently with the main processor's instruction execution. If the coprocessor no longer requires the services of the main processor to complete a cpGEN instruction and the concurrent instruction completion is transparent to the programmer's model, the coprocessor can release the main processor by issuing a primitive with CA = 0. The main processor usually executes the next instruction in the instruction stream, and the coprocessor completes its operations concurrently with the main processor operation. If an exception occurs while the coprocessor is executing an instruction concurrently, the exception is not processed until the main processor attempts to initiate the next general or conditional instruction. After the main processor writes to the command or condition CIR to initiate a general or conditional instruction, it then reads the response CIR. At this time, the coprocessor can return the take preinstruction exception primitive. This protocol allows the main processor to proceed with exception processing related to the previous concurrently executing coprocessor instruction and then return and reinitiate the coprocessor instruction during which the exception was signaled. The coprocessor should record the addresses of all general category instructions that can be executed concurrently with the main processor and that support exception recovery. Since the exception is not reported until the next coprocessor instruction is initiated, the processor usually requires the instruction address to determine which instruction the coprocessor was executing when the exception occurred. A coprocessor can record the instruction address by setting PC = 1 in one of the primitives it uses before releasing the main processor.

9.4.19 Take Midinstruction Exception Primitive

The take midinstruction exception primitive initiates exception processing using a coprocessor supplied exception vector number and the midinstruction exception stack frame format. This primitive applies to general and conditional category instructions. Figure 9-42 shows the format of the take midinstruction exception primitive.

This primitive uses the PC bit as previously described. Bits [7–0] contain the exception vector number used by the main processor to initiate exception processing.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	PC	0	1	1	1	0	1	VECTOR NUMBER							

Figure 9-42. Take Midinstruction Exception Primitive Format

When the main processor receives this primitive, it acknowledges the coprocessor exception request by writing an exception acknowledge mask (refer to **9.3.2 Control CIR**) to the control CIR. The MC68020 then performs exception processing as described in **8.1 EXCEPTION PROCESSING SEQUENCE**. The vector number for the exception is taken from bits [7–0] of the primitive and the MC68020 uses the ten-word stack frame format shown in Figure 9-43.

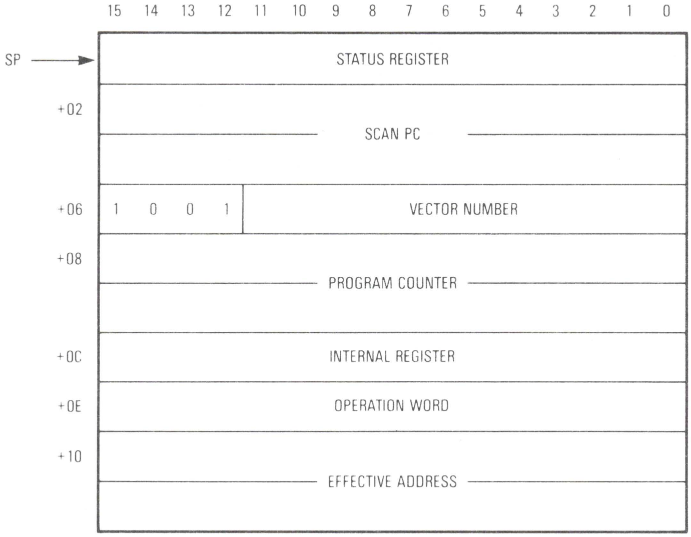


Figure 9-43. MC68020 Midinstruction Stack Frame

The program counter value saved in this stack frame is the operation word address of the coprocessor instruction during which the primitive is received. The scanPC field contains the value of the MC68020 scanPC when the primitive is received. If the current instruction does not evaluate an effective address prior to the exception request primitive, the value of the effective address field in the stack frame is undefined.

The coprocessor uses this primitive to request exception processing for an exception during the instruction dialog with the main processor. If the exception handler does not modify the stack frame, the MC68020 returns from the exception handler and reads the response CIR. Thus, the main processor attempts to continue executing the suspended instruction by reading the response CIR and processing the primitive it receives.

9.4.20 Take Postinstruction Exception Primitive

The take postinstruction exception primitive initiates exception processing using a coprocessor-supplied exception vector number and the postinstruction exception stack frame format. This primitive applies to general and conditional category instructions. Figure 9-44 shows the format of the take postinstruction exception primitive.

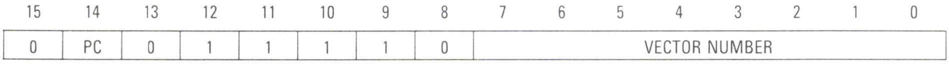


Figure 9-44. Take Postinstruction Exception Primitive Format

This primitive uses the PC bit as previously described. Bits [7–0] contain the exception vector number used by the main processor to initiate exception processing.

When the main processor receives this primitive, it acknowledges the coprocessor exception request by writing an exception acknowledge mask (refer to **9.3.2 Control CIR**) to the control CIR. The MC68020 then performs exception processing as described in **8.1 EXCEPTION PROCESSING SEQUENCE**. The vector number for the exception is taken from bits [7–0] of the primitive and the MC68020 uses the six-word stack frame format shown in Figure 9-45.

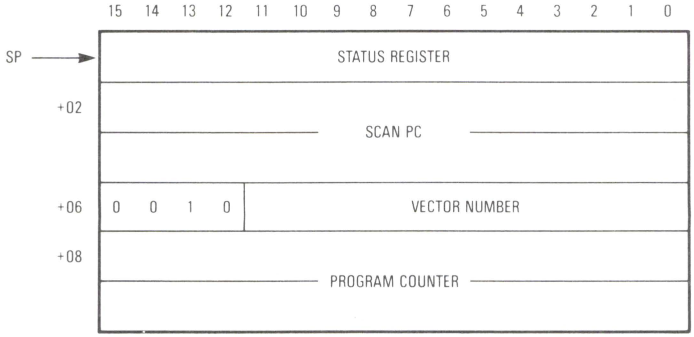


Figure 9-45. MC68020 Postinstruction Stack Frame

The value in the main processor scanPC at the time this primitive is received is saved in the scanPC field of the postinstruction exception stack frame. The value of the program counter saved is the F-line operation word address of the coprocessor instruction during which the primitive is received.

When the MC68020 receives the take postinstruction exception primitive, it assumes that the coprocessor either completed or aborted the instruction with an exception. If the exception handler does not modify the stack frame, the MC68020 returns from the exception handler to begin execution at the location specified by the scanPC field of the stack frame. This location should be the address of the next instruction to be executed.

The coprocessor uses this primitive to request exception processing when it completes or aborts an instruction while the main processor is awaiting a normal response. For a general category instruction, the response is a release; for a conditional category instruction, it is an evaluated true/false condition indicator. Thus, the operation of the MC68020 in response to this primitive is compatible with standard M68000 Family instruction related exception processing (for example, the divide-by-zero exception).

9.5 EXCEPTIONS

Various exception conditions related to the execution of coprocessor instructions may occur. Whether an exception is detected by the main processor or by the coprocessor, the main processor coordinates and performs exception processing. Servicing these coprocessor related exceptions is an extension of the protocol used to service standard M68000 Family exceptions. That is, when either the main processor detects an exception or is signaled by the coprocessor that an exception condition has occurred, the main processor proceeds with exception processing as described in **8.1 EXCEPTION PROCESSING SEQUENCE**.

9.5.1 Coprocessor Detected Exceptions

Exceptions that the coprocessor detects, also those that the main processor detects, are usually classified as coprocessor detected exceptions. These exceptions can occur during M68000 coprocessor interface operations, internal operations, or other system-related operations of the coprocessor.

Most coprocessor detected exceptions are signaled to the main processor through the use of one of the three take exception primitives defined for the M68000 coprocessor interface. The main processor responds to these primitives as previously described. However, not all coprocessor detected exceptions are signaled by response primitives. Coprocessor detected format errors during the cpSAVE or cpRESTORE instruction are signaled to the main processor using the invalid format word described in **9.2.3.4.3 Invalid Format Words**.

9.5.1.1 COPROCESSOR DETECTED PROTOCOL VIOLATIONS. Protocol violation exceptions are communication failures between the main processor and coprocessor across the M68000 coprocessor interface. Coprocessor detected protocol violations occur when the main processor accesses entries in the coprocessor interface register set in an unexpected sequence. The sequence of operations that the main processor performs for a given coprocessor instruction or coprocessor response primitive has been described previously in this section.

A coprocessor can detect protocol violations in various ways. According to the M68000 coprocessor interface protocol, the main processor always accesses the operation word, operand, register select, instruction address, or operand address CIRs synchronously with respect to the operation of the coprocessor. That is, the main processor accesses these five registers in a certain sequence, and the coprocessor expects them to be accessed in that sequence. As a minimum, all M68000 coprocessors should detect a protocol violation if the main processor accesses any of these five registers when the coprocessor is expecting an access to either the command or condition CIR. Likewise, if the coprocessor is expecting an access to the command or condition CIR and the main processor accesses one of these five registers, the coprocessor should detect and signal a protocol violation.

According to the M68000 coprocessor interface protocol, the main processor can perform a read of either the save or response CIRs or a write of either the restore or control CIRs asynchronously with respect to the operation of the coprocessor. That is, an access to one of these registers without the coprocessor explicitly expecting that access at that point can be a valid access. Although the coprocessor can anticipate certain accesses to the restore, response, and control coprocessor interface registers, these registers can be accessed at other times also.

The coprocessor cannot signal a protocol violation to the main processor during the execution of cpSAVE or cpRESTORE instructions. If a coprocessor detects a protocol violation during the cpSAVE or cpRESTORE instruction, it should signal the exception to the main processor when the next coprocessor instruction is initiated.

The main philosophy of the coprocessor detected protocol violation is that the coprocessor should always acknowledge an access to one of its interface registers. If the coprocessor determines that the access is not valid, it should assert $\overline{\text{DSACKx}}$ to the main processor and signal a protocol violation when the main processor next reads the response CIR. If the coprocessor fails to assert $\overline{\text{DSACKx}}$ the main processor waits for the assertion of that signal (or some other bus termination signal) indefinitely. The protocol previously described ensures that the coprocessor cannot halt the main processor.

The coprocessor can signal a protocol violation to the main processor with the take midinstruction exception primitive. To maintain consistency, the vector number should be 13, as it is for a protocol violation detected by the main processor. When the main processor reads this primitive, it proceeds as described in **9.4.19 Take Midinstruction Exception Primitive**. If the exception handler does not modify the stack frame, the MC68020 returns from the exception handler and reads the response CIR.

9.5.1.2 COPROCESSOR DETECTED ILLEGAL COMMAND OR CONDITION WORDS. Illegal coprocessor command or condition words are values written to the command CIR or condition CIR that the coprocessor does not recognize. If a value written to either of these registers is not valid, the coprocessor should return the take preinstruction exception primitive in the response CIR. When it receives this primitive, the main processor takes a preinstruction exception as described in **9.4.18 Take Preinstruction Exception Primitive**. If the exception handler does not modify the main processor stack frame, an RTE instruction causes the MC68020 to reinitiate the instruction that took the exception. The coprocessor designer should ensure that the state of the coprocessor is not irrecoverably altered by an illegal command or condition exception if the system supports emulation of the unrecognized command or condition word.

All Motorola M68000 coprocessors signal illegal command and condition words by returning the take pre-instruction exception primitive with the F-line emulator exception vector number 11.

9.5.1.3 COPROCESSOR DATA PROCESSING EXCEPTIONS. Exceptions related to the internal operation of a coprocessor are classified as data processing related exceptions. These exceptions are analogous to the divide-by-zero exception defined by M68000 microprocessors and should be signaled to the main processor using one of the three take exception primitives containing an appropriate exception vector number. Which of these three primitives is used to signal the exception is usually determined by the point in the instruction operation where the main processor should continue the program flow after exception processing. Refer to **9.4.18 Take Preinstruction Exception Primitives**, **9.4.19 Take Midinstruction Exception Primitive**, and **9.4.20 Take Postinstruction Exception Primitive**.

9.5.1.4 COPROCESSOR SYSTEM-RELATED EXCEPTIONS. System-related exceptions detected by a DMA coprocessor include those associated with bus activity, and any other exceptions (interrupts, for example) occurring external to the coprocessor. The actions taken by the coprocessor and the main processor depend on the type of exception that occurs.

When an address or bus error is detected by a DMA coprocessor, the coprocessor should store any information necessary for the main processor exception handling routines in system accessible registers. The coprocessor should place one of the three take exception primitives encoded with an appropriate exception vector number in the response CIR. Which of the three primitives is used depends upon the point in the coprocessor instruction at which the exception was detected, and the point in the instruction execution at which the main processor should continue after exception processing.

9.5.1.5 FORMAT ERRORS. Format errors are the only coprocessor detected exceptions that are not signaled to the main processor with a response primitive. When the main processor writes a format word to the restore CIR during the execution of a cpRESTORE instruction, the coprocessor decodes this word to determine if it is valid (refer to **9.2.3.3 COPROCESSOR CONTEXT SAVE INSTRUCTION**). If the format word is not valid, the coprocessor places the invalid format code in the restore CIR. When the main processor reads the invalid format code, it aborts the coprocessor instruction by writing an abort mask (refer to **9.3.2 Control CIR**) to the control CIR. The main processor then performs exception processing using a four-word preinstruction stack frame and the format error exception vector number 14. Thus, if the exception handler does not modify the stack frame, the MC68020 restarts the cpRESTORE instruction when the RTE instruction in the handler is executed. If the coprocessor returns the invalid format code when the main processor reads the save CIR to initiate a cpSAVE instruction, the main processor performs format error exception processing as outlined for the cpRESTORE instruction.

9.5.2 Main Processor Detected Exceptions

A number of exceptions related to coprocessor instruction execution are detected by the main processor instead of the coprocessor (they are still serviced by the main processor). These exceptions can be related to the execution of coprocessor response primitives, communication across the M68000 coprocessor interface, or the completion of conditional coprocessor instructions by the main processor.

9.5.2.1 PROTOCOL VIOLATIONS. The main processor detects a protocol violation when it reads a primitive from the response CIR that is not a valid primitive. The protocol violations that can occur in response to the primitives defined for the M68000 coprocessor interface are summarized in Table 9-6.

Table 9-6. Exceptions Related to Primitive Processing

Primitive	Protocol	F-Line	Other
Busy			
NULL			
Supervisory Check* Other: Privilege Violation if "S" Bit = 0			X
Transfer Operation Word*			
Transfer from Instruction Stream* Protocol: If Length Field is Odd (Zero Length Legal)	X		
Evaluate and Transfer Effective Address Protocol: If Used with Conditional Instruction F-Line: If EA in Op-Word is NOT Control Alterable	X	X	
Evaluate Effective Address and Transfer Data Protocol: 1. If Used with Conditional Instructions 2. Length is Not 1, 2, or 4 ad EA = Register Direct 3. If EA = Immediate and Length Odd and Greater Than 1 4. Attempt to Write to Non-Alterable Address Even if Address Declared Legal in Primitive F-Line: Valid EA Field Does Not Match EA in Op-Word	X	X	
Write to Previously Evaluated Effective Address Protocol: If Used with Conditional Instruction	X		
Busy			
Take Address and Transfer Data*			
Transfer To/From Top-of-Stack* Protocol: Length Field Other Than 1, 2, or 4	X		
Transfer To/From Main Processor Register*			
Transfer To/From Main Processor Control Register Protocol: Invalid Control Register Select Code	X		
Transfer Multiple Main Processor Registers*			
Transfer Multiple Coprocessor Registers Protocol: 1. If Used with Conditional Instructions 2. Odd Length Value F-Line: 1. EA Not Control Alterable or (An)+ for CP to Memory Transfer 2. EA Not Control Alterable or −(An) for Memory to CP Transfer	X X		
Transfer Status and/or ScanPC Protocol: If Used with Conditional Instruction Other: 1. Trace — Trace Made Pending if MC68020 in "Trace on Change of Flow" Mode and DR = 1 2. Address Error — If Odd Value Written to ScanPC	X		X
Take Preinstruction, Midinstruction, or Postinstruction Exception Exception Depends on Vector Supplies in Primitive	X	X	X

*Use of this primitive with CA = 0 will cause protocol violation on conditional instructions.

Abbreviations:
EA = Effective Address
CP = Coprocessor

When the MC68020 detects a protocol violation, it does not automatically notify the coprocessor of the resulting exception by writing to the control CIR. The exception handling routine may, however, use the MOVES instruction to read the response CIR and thus determine the primitive that caused the MC68020 to initiate protocol violation exception processing. The main processor initiates exception processing using the midinstruction stack frame (refer to Figure 9-43) and the coprocessor protocol violation exception vector number 13. If the exception handler does not modify the stack frame, the main processor reads the response CIR again following the execution of an RTE instruction to return from the exception handler. This protocol allows extensions to the M68000 coprocessor interface to be emulated in software by a main processor that does not provide hardware support for these extensions. Thus, the protocol violation is transparent to the coprocessor if the primitive execution can be emulated in software by the main processor.

9.5.2.2 F-LINE EMULATOR EXCEPTIONS. The F-line emulator exceptions detected by the MC68020 are either explicitly or implicitly related to the encodings of F-line operation words in the instruction stream. If the main processor determines that an F-line operation word is not valid, it initiates F-line emulator exception processing. Any F-line operation word with bits [8–6] = 110 or 111 causes the MC68020 to initiate exception processing without initiating any communication with the coprocessor for that instruction. Also, an operation word with bits [8–6] = 000–101 that does not map to one of the valid coprocessor instructions in the instruction set (**SECTION 3 INSTRUCTION SET SUMMARY**) causes the MC68020 to initiate F-line emulator exception processing. If the F-line emulator exception is either of these two situations, the main processor does not write to the control CIR prior to initiating exception processing.

F-line exceptions can also occur if the operations requested by a coprocessor response primitive are not compatible with the effective address type in bits [5–0] of the coprocessor instruction operation word. The F-line emulator exceptions that can result from the use of the M68000 coprocessor response primitives are summarized in Table 9-6. If the exception is caused by receiving an invalid primitive, the main processor aborts the coprocessor instruction in progress by writing an abort mask (refer to **9.3.2 Control CIR**) to the control CIR prior to F-line emulator exception processing.

Another type of F-line emulator exception occurs when a bus error occurs during the coprocessor interface register access that initiates a coprocessor instruction. The main processor assumes that the coprocessor is not present and takes the exception.

When the main processor initiates F-line emulator exception processing, it uses the four-word preinstruction exception stack frame (refer to Figure 9-41) and the F-line emulator exception vector number 11. Thus, if the exception handler does not modify the stack frame, the main processor attempts to restart the instruction that caused the exception after it executes an RTE instruction to return from the exception handler.

If the cause of the F-line exception can be emulated in software, the handler stores the results of the emulation in the appropriate registers of the programmer's model and in the status register field of the saved stack frame. The exception handler adjusts the program counter field of the saved stack frame to point to the next instruction operation word and executes the RTE instruction. The MC68020 then executes the instruction following the instruction that was emulated.

The exception handler should also check the copy of the status register on the stack to determine whether tracing is on. If tracing is on, the trace exception processing should also be emulated. Refer to **8.1.7 Trace Exception** for additional information.

9.5.2.3 PRIVILEGE VIOLATIONS.

Privilege violations can result from the cpSAVE and cpRESTORE instructions and also from the supervisor check coprocessor response primitive. The main processor initiates privilege violation exception processing if it attempts to execute either the cpSAVE or cpRESTORE instruction when it is in the user state ($S=0$ in status register). The main processor initiates this exception processing prior to any communication with the coprocessor associated with the cpSAVE or cpRESTORE instructions.

If the main processor is executing a coprocessor instruction in the user state when it reads the supervisor check primitive, it aborts the coprocessor instruction in progress by writing an abort mask (refer to **9.3.2 Control CIR**) to the control CIR. The main processor then performs privilege violation exception processing.

If a privilege violation occurs, the main processor initiates exception processing using the four-word preinstruction stack frame (refer to Figure 9-41) and the privilege violation exception vector number 8. Thus, if the exception handler does not modify the stack frame, the main processor attempts to restart the instruction during which the exception occurred after it executes an RTE to return from the handler.

9.5.2.4 cpTRAPcc INSTRUCTION TRAPS.

If, during the execution of a cpTRAPcc instruction, the coprocessor returns the TRUE condition indicator to the main processor with a null primitive, the main processor initiates trap exception processing. The main processor uses the six-word postinstruction exception stack frame (refer to Figure 9-45) and the trap exception vector number 7. The scanPC field of this stack frame contains the address of the instruction following the cpTRAPcc instruction. The processing associated with the cpTRAPcc instruction can then proceed, and the exception handler can locate any immediate operand words encoded in the cpTRAPcc instruction using the information contained in the six-word stack frame. If the exception handler does not modify the stack frame, the main processor executes the instruction following the cpTRAPcc instruction after it executes an RTE instruction to exit from the handler.

9.5.2.5 TRACE EXCEPTIONS.

The MC68020 supports two modes of instruction tracing, discussed in **8.1.7 Trace Exception**. In the trace on instruction execution mode, the MC68020 takes a trace exception after completing each instruction. In the trace on change of flow mode, the MC68020 takes a trace exception after each instruction that alters the status register or places an address other than the address of the next instruction in program counter.

The protocol used to execute coprocessor cpSAVE, cpRESTore, or conditional category instructions does not change when a trace exception is pending in the main processor. The main processor performs a pending trace on instruction execution exception after completing the execution of that instruction. If the main processor is in the trace on change of flow mode, and an instruction places an address other than that of the next instruction in the program counter, the processor takes a trace exception after it executes the instruction.

If a trace exception is not pending during a general category instruction, the main processor terminates communication with the coprocessor after reading any primitive with $CA=0$.

Thus, the coprocessor can complete a cpGEN instruction concurrently with the execution of instructions by the main processor. When a trace exception is pending, however, the main processor must ensure that all processing associated with a cpGEN instruction has been completed before it takes the trace exception. In this case, the main processor continues to read the response CIR and service the primitives until it receives either a null, CA = 0, PF = 1 primitive or until exception processing caused by a take postinstruction exception primitive has completed. The coprocessor should return the null, CA = 0 primitive with PF = 0 while it is completing the execution of the cpGEN instruction. The main processor may service pending interrupts between reads of the response CIR if IA = 1 in these primitives (refer to Table 9-3). This protocol ensures that a trace exception is not taken until all processing associated with a cpGEN instruction has completed.

If T1:T0 = 01 in the MC68020 status register (trace on change of flow) when a general category instruction is initiated, a trace exception is taken for the instruction only when the coprocessor issues a transfer status register and scanPC primitive with DR = 1 during the execution of that instruction. In this case, it is possible that the coprocessor is still executing the cpGEN instruction concurrently when the main processor begins execution of the trace exception handler. A cpSAVE instruction executed during the trace on change of flow exception handler could thus suspend the execution of a concurrently operating cpGEN instruction.

9.5.2.6 INTERRUPTS. Interrupt processing, discussed in **8.1.9 Interrupt Exceptions**, can occur at any instruction boundary. Interrupts are also serviced during the execution of a general or conditional category instruction under either of two conditions. If the main processor reads a null primitive with CA = 1 and IA = 1, it services any pending interrupts prior to reading the response CIR. Similarly, if a trace exception is pending during cpGEN instruction execution and the main processor reads a null primitive with CA = 0, IA = 1, and PF = 0 (refer to **9.5.2.5 TRACE EXCEPTIONS**), the main processor services pending interrupts prior to reading the response CIR again.

The MC68020 uses the ten-word midinstruction stack frame when it services interrupts during the execution of a general or conditional category coprocessor instruction. Since it uses this stack frame, the main processor can perform all necessary processing and then return to read the response CIR. Thus, it can continue the coprocessor instruction during which the interrupt exception was taken.

The MC68020 also services interrupts if it reads the not-ready format word from the save CIR during a cpSAVE instruction. The MC68020 uses the normal four-word preinstruction stack frame when it services interrupts after reading the not-ready format word. Thus, the processor can service any pending interrupts and execute an RTE to return and reinitiate the cpSAVE instruction by reading the save CIR.

9.5.2.7 MAIN PROCESSOR DETECTED FORMAT ERRORS. The MC68020 can detect a format error while executing a cpSAVE or cpRESTORE instruction if the length field of a valid format word is not a multiple of four bytes in length. If the MC68020 reads a format word with an invalid length field from the save CIR during the cpSAVE instruction, it aborts the coprocessor instruction by writing an abort mask (refer to **9.3.2 Control CIR**) to the control

CIR and initiates format error exception processing. If the MC68020 reads a format word with an invalid length field from the effective address specified in the cpRESTORE instruction, the MC68020 writes that format word to the restore CIR and then reads the coprocessor response from the restore CIR. The MC68020 then aborts the cpRESTORE instruction by writing an abort mask (refer to **9.3.2 Control CIR**) to the control CIR and initiates format error exception processing.

The MC68020 uses the four-word preinstruction stack frame and the format error vector number 14 when it initiates format error exception processing. Thus, if the exception handler does not modify the stack frame, the main processor attempts to restart the instruction during which the exception occurred after it executes an RTE to return from the handler.

9.5.2.8 ADDRESS AND BUS ERRORS. Coprocessor instruction related bus faults can occur during main processor bus cycles to CPU space to communicate with a coprocessor or during memory cycles run as part of the coprocessor instruction execution. If a bus error occurs during the coprocessor interface register access that is used to initiate a coprocessor instruction, the main processor assumes that the coprocessor is not present and takes an F-line emulator exception as described in **9.5.2.2 F-LINE EMULATOR EXCEPTIONS**. That is, the processor takes an F-line emulator exception when a bus error occurs during the initial access to a CIR by a coprocessor instruction. If a bus error occurs on any other coprocessor access or on a memory access made during the execution of a coprocessor instruction, the main processor performs bus error exception processing as described in **8.1.2 Bus Error Exceptions**. After the exception handler has corrected the cause of the bus error, the main processor can return to the point in the coprocessor instruction at which the fault occurred.

An address error occurs if the MC68020 attempts to prefetch an instruction from an odd address. This can occur if the calculated destination address of a cpBcc or cpDBcc instruction is odd, or if an odd value is transferred to the scanPC with the transfer status register and the scanPC response primitive. If an address error occurs, the MC68020 performs exception processing for a bus fault as described in **8.1.3 Address Error Exception**.

9.5.3 Coprocessor Reset

Either an external reset signal or a RESET instruction can reset the external devices of a system. The system designer can design a coprocessor to be reset and initialized by both reset types or by external reset signals only. To be consistent with the MC68020 design, the coprocessor should be affected by external reset signals only and not by RESET instructions, because the coprocessor is an extension to the main processor programming model and to the internal state of the MC68020.

9.6 COPROCESSOR SUMMARY

Coprocessor instruction formats are included with the instruction formats in **3.9 INSTRUCTION FORMAT SUMMARY**.

The M68000 coprocessor response primitive formats are shown in this section. Any response primitive with bits [13–8] = $00 or $3F causes a protocol violation exception. Response primitives with bits [13–8] = $0B, $18-$1B, $1F, $28-$2B, and $38-3B currently cause protocol violation exceptions, but are undefined and reserved for future use by Motorola.

BUSY

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	PC	1	0	0	1	0	0	0	0	0	0	0	0	0	0

TRANSFER MULTIPLE COPROCESSOR REGISTERS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	0	0	1	LENGTH							

TRANSFER STATUS REGISTER AND SCANPC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	0	1	SP	0	0	0	0	0	0	0	0

SUPERVISOR CHECK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	PC	0	0	0	1	0	0	0	0	0	0	0	0	0	0

TAKE ADDRESS AND TRANSFER DATA

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	1	0	1	LENGTH							

TRANSFER MULTIPLE MAIN PROCESSOR REGISTERS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	1	1	0	0	0	0	0	0	0	0	0

TRANSFER OPERATION WORD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	0	1	1	1	0	0	0	0	0	0	0	0

NULL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	1	0	0	IA	0	0	0	0	0	0	PF	TF

EVALUATE AND TRANSFER EFFECTIVE ADDRESS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	1	0	1	0	0	0	0	0	0	0	0	0

TRANSFER SINGLE MAIN PROCESSOR REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	1	1	0	0	0	0	0	0	D/A	REGISTER		

TRANSFER MAIN PROCESSOR CONTROL REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	1	1	0	1	0	0	0	0	0	0	0	0

TRANSFER TO/FROM TOP OF STACK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	1	1	1	0	LENGTH							

TRANSFER FROM INSTRUCTION STREAM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	1	1	1	1	LENGTH							

EVALUATE EFFECTIVE ADDRESS AND TRANSFER DATA

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	1	0	VALID EA			LENGTH							

TAKE PREINSTRUCTION EXCEPTION

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	PC	0	1	1	1	0	0	VECTOR NUMBER							

TAKE MIDINSTRUCTION EXCEPTION

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	PC	0	1	1	1	0	1	VECTOR NUMBER							

TAKE POSTINSTRUCTION EXCEPTION

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	PC	0	1	1	1	1	0	VECTOR NUMBER							

WRITE TO PREVIOUSLY EVALUATED EFFECTIVE ADDRESS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	1	0	0	0	0	0	LENGTH							

9

SECTION 10
INSTRUCTION EXECUTION TIMING

This section describes the instruction execution and operations (table searches, etc.) of the MC68020 in terms of external clock cycles. It provides accurate execution and operation timing guidelines but not exact timings for every possible circumstance. This approach is used since exact execution time for an instruction or operation is highly dependent on memory speeds and other variables. The timing numbers presented in this section allow the assembly-language programmer or compiler writer to predict timings needed to evaluate the performance of the MC68020.

In this section, instruction and operation times are shown in clock cycles in order to eliminate clock frequency dependencies.

10.1 TIMING ESTIMATION FACTORS

The advanced architecture of the MC68020 makes exact instuction timing calculations difficult due to the effects of:

1) An On-Chip Instruction Cache and Instruction Prefetch
2) Operand Misalignment
3) Bus Controller/Sequence Concurrency
4) Instruction Execution Overlap

These factors make MC68020 instruction set timing difficult to calculate on a single instruction basis since instructions vary in execution time from one context to another. A detailed explanation of each of these factors follows.

10.1.1 Instruction Cache and Prefetch

The on-chip cache of the MC68020 is an instruction-only cache. Its purpose is to increase execution efficiency by providing a quick-store for instructions.

Instruction prefetches that hit in the cache will occur with no delay in instruction execution. Instruction prefetches that miss in the cache will cause an external memory cycle to be performed, which may overlap with internal instruction execution. Thus, while the execution unit of the microprocessor is busy, the bus controller prefetches the next instruction from external memory. Both cases are illustrated in later examples.

When prefetching instructions from extenal memory, the microprocessor will utilize long-word read cycles. When the read is aligned on a long-word address boundary, the processor reads two words, which may load two instructions at once, or two words of a multiword instruction. The subsequent instruction prefetch will find the second word is already available, and there is no need to run an external bus cycle (read).

The MC68020 always prefetches long words. When an instruction prefetch falls on an odd-word boundary (e.g., due to a branch to an odd-word location), the MC68020 will read the even word associated with the long-word base address at the same time as (32-bit memory) or before (8- or 16-bit memory) the odd word is read. When an instruction prefetch falls on an even-word boundary (as would be the normal case), the MC68020 reads both words at the long-word address, thus effectively prefetching the next two words.

10.1.2 Operand Misalignment

Another significant factor affecting instruction timing is operand misalignment. Operand misalignment has impact on performance when the microprocessor is reading or writing external memory. In this case, the address of a word operand falls across a long-word boundary or a long-word operand falls on a byte or word address which is not a long-word boundary. While the MC68020 will automatically handle all occurrences of operand misalignment, it must use multiple bus cycles to complete such transfers.

10.1.3 Bus/Sequencer Concurrency

The bus controller is responsible for all bus activity. The sequencer controls the bus controller, instruction execution, and internal processor operation, such as calculation of effective addresses and setting of condition codes.

The bus controller and sequencer can operate on an instruction concurrently. The bus controller can perform a read or write while the sequencer controls an effective address calculation or sets the condition codes. The sequencer may also request a bus cycle that the bus controller cannot immediately perform. In this case, the bus cycle is queued and the bus controller runs the cycle when the current cycle is complete.

10.1.4 Instruction Execution Overlap

Overlap is the time, measured in clocks, when two instructions execute concurrently. In Figure 10-1, instructions A and B execute concurrently, and the overlapped portion of instruction B is absorbed in the instruction execution time of A (the previous instruction). The overlap time is deducted from the execution time of instruction B. Similarly, there is an overlap period between instruction B and instruction C, which reduces the attributed execution time for C.

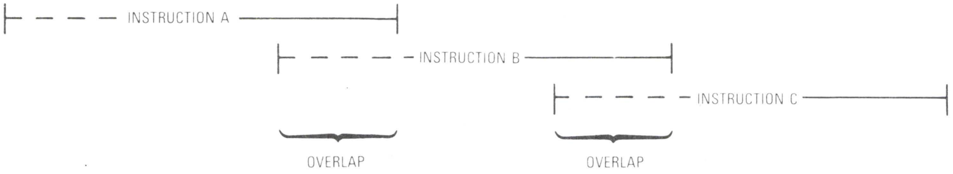


Figure 10-1. Concurrent Instruction Execution

The execution time attributed to instructions A, B, and C (after considering the overlap) is depicted in Figure 10-2.

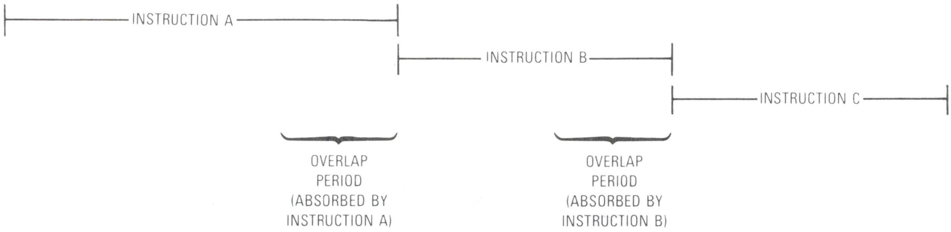

INSTRUCTION A
INSTRUCTION B
INSTRUCTION C
OVERLAP PERIOD (ABSORBED BY INSTRUCTION A)
OVERLAP PERIOD (ABSORBED BY INSTRUCTION B)

Figure 10-2. Instruction Execution for Instruction Timing Purposes

It is possible that the execution time of an instruction will be absorbed by the overlap with a previous instruction for a net execution time of zero clocks.

Because of this overlap, a NOP is required between a write to a peripheral to clear an interrupt request, and a subsequent move to SR to lower the interrupt mask level. Otherwise, the move to SR may complete before the write is accomplished, and a new interrupt exception will be generated for an old interrupt request.

10.1.5 Instruction Stream Timing Examples

A programming example allows a more detailed examination of these effects. The effect of instruction execution overlap on instruction timing is illustrated by the following example instruction stream:

Instruction

#1) MOVE.L D4,(A1)+
#2) ADD.L D4,D5
#3) MOVE.L (A1),−(A2)
#4) ADD.L D5,D6

For the first example, the assumptions are:

1) The data bus is 32 bits,
2) The first instruction is prefetched from an ODD word address,
3) Memory access with no wait states, and
4) The instruction cache is disabled.

For this example, the instruction stream is positioned in 32-bit memory as:

Address n	• • •	MOVE #1
n+4	ADD #2	MOVE #3
n+8	ADD #4	• • •

Figure 10-3 shows processor activity on the first example instruction stream. It shows the activity of the external bus, the bus controller, the sequencer, and the attributed instruction execution time.

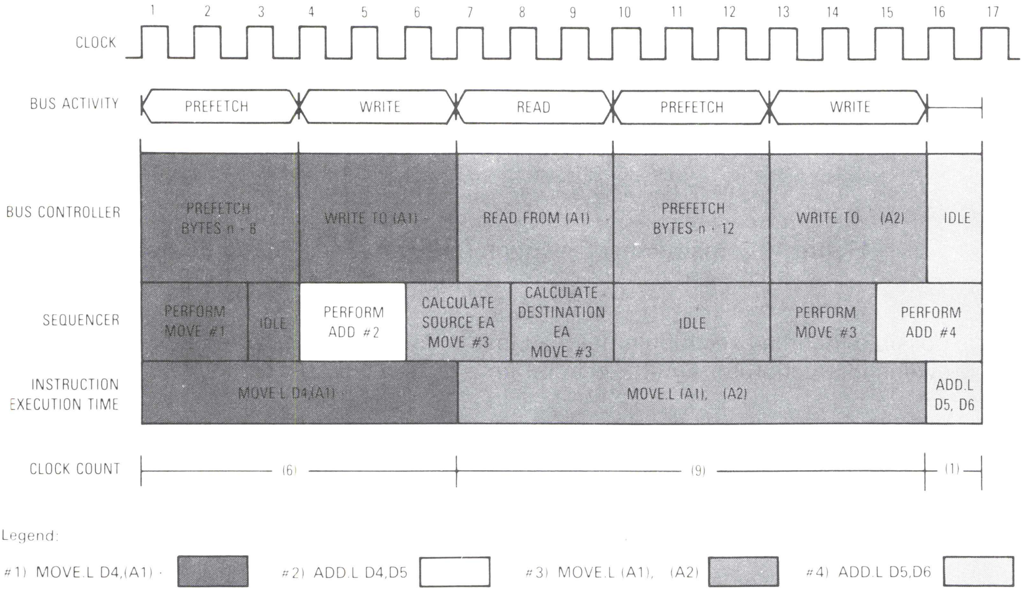

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
CLOCK
BUS ACTIVITY
PREFETCH
WRITE
READ
PREFETCH
WRITE
BUS CONTROLLER
PREFETCH BYTES n + 8
WRITE TO (A1)
READ FROM (A1)
PREFETCH BYTES n + 12
WRITE TO (A2)
IDLE
SEQUENCER
PERFORM MOVE #1
IDLE
PERFORM ADD #2
CALCULATE SOURCE EA MOVE #3
CALCULATE DESTINATION EA MOVE #3
IDLE
PERFORM MOVE #3
PERFORM ADD #4
INSTRUCTION EXECUTION TIME
MOVE.L D4,(A1)
MOVE.L (A1), (A2)
ADD.L D5, D6
CLOCK COUNT
(6)
(9)
(1)
Legend:
#1) MOVE.L D4,(A1)
#2) ADD.L D4,D5
#3) MOVE.L (A1), (A2)
#4) ADD.L D5,D6

Figure 10-3. Processor Activity Example 1

For the first three clocks of this example, the bus controller and sequencer are both performing tasks associated with the MOVE #1 instruction. The next three clocks (clocks four, five, and six) demonstrate instruction overlap. The bus controller is performing a write to memory as part of the MOVE #1 instruction. The sequencer, on the other hand, is performing the ADD #2 instruction for two clocks (clocks four and five) and beginning source effective address (EA) calculations for the MOVE #3 instruction. The bus controller activity completely overlaps the execution of the ADD #2 instruction, causing the ADD #2 attributed execution time to be zero clocks. The overlap also shortens the effective execution time of the MOVE #3 instruction by the one clock because the bus controller completes the MOVE #1 write operation while the sequencer begins the MOVE #3 effective address calculation.

The sequencer continues the source EA calculation for one more clock period (clock seven) while the bus controller begins a read for MOVE #3. When counting instruction execution time in bus clocks, the MOVE #1 completes at the end of clock 6, and the execution of MOVE #3 begins on clock 7.

Both the sequencer and bus controller continue with MOVE #3 until the end of clock 14, when the sequencer begins to perform ADD #4. Timing for MOVE #3 continues, because the bus controller is still performing the write to the destination of MOVE #3. The bus activity for MOVE #3 completes at the end of clock 15. The effective execution time for MOVE #3 is nine clocks.

The one clock cycle (clock 15) when the sequencer is performing ADD #4 and the bus controller is writing to the destination of MOVE #3 is absorbed by the execution time of MOVE #3. This shortens the effective execution time of ADD #4 by one clock, giving it an attributed execution time of one clock.

Using the same instruction stream, the second example demonstrates the different effects of instruction execution overlap on instruction timing when the same instructions are positioned slightly differently in 32-bit memory:

Address			
	n	MOVE #1	ADD #2
	n+4	MOVE #3	ADD #4
	n+8	• • •	• • •

The assumptions for the second example in Figure 10-4 are:
1) The data bus is 32 bits,
2) The first instruction is prefetched from an EVEN word address,
3) Memory access occurs with no wait states, and
4) The cache is disabled.

While the total execution time of the instruction segment does not change in this example, the individual instruction times are significantly different. This demonstrates that the effects

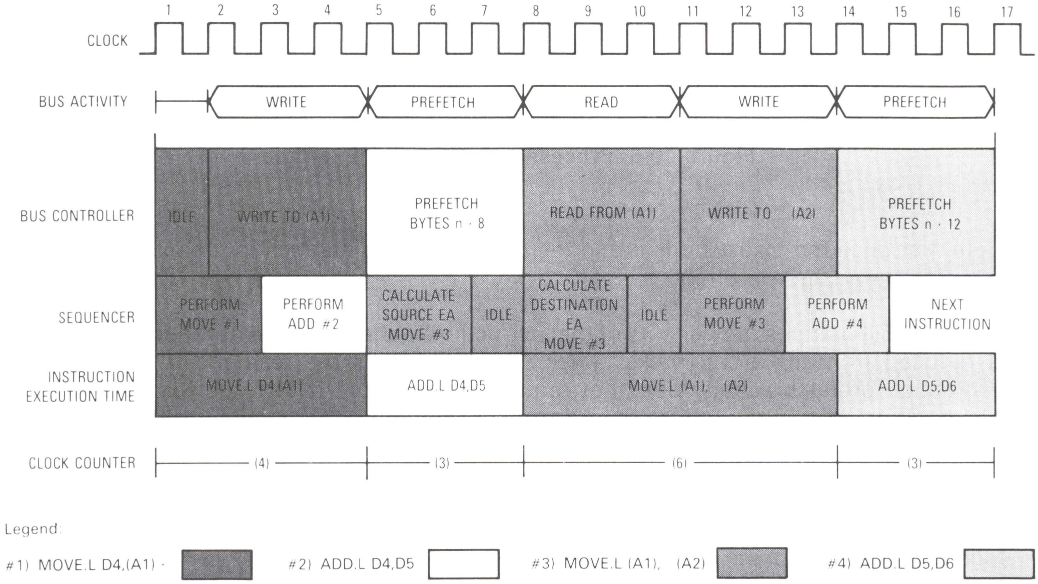

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
CLOCK
BUS ACTIVITY
WRITE
PREFETCH
READ
WRITE
PREFETCH
BUS CONTROLLER
IDLE
WRITE TO (A1)+
PREFETCH BYTES n + 8
READ FROM (A1)
WRITE TO (A2)
PREFETCH BYTES n + 12
SEQUENCER
PERFORM MOVE #1
PERFORM ADD #2
CALCULATE SOURCE EA MOVE #3
IDLE
CALCULATE DESTINATION EA MOVE #3
IDLE
PERFORM MOVE #3
PERFORM ADD #4
NEXT INSTRUCTION
INSTRUCTION EXECUTION TIME
MOVE.L D4,(A1)
ADD.L D4,D5
MOVE.L (A1), (A2)
ADD.L D5,D6
CLOCK COUNTER
(4)
(3)
(6)
(3)
Legend:
#1) MOVE.L D4,(A1)+
#2) ADD.L D4,D5
#3) MOVE.L (A1), (A2)
#4) ADD.L D5,D6

Figure 10-4. Processor Activity for Example 2

of overlap are not only instruction-sequence dependent but are also dependent upon the alignment of the instruction stream in memory.

Both Figures 10-3 and 10-4 show instruction execution without benefit of the MC68020 instruction cache. Figure 10-5 shows a third example for the same instruction stream executing in the cache. The assumptions for Example 3 are:

1) The data bus is 32 bits,
2) The cache is enabled and instructions are in the cache, and
3) Memory access occurs with no wait states.

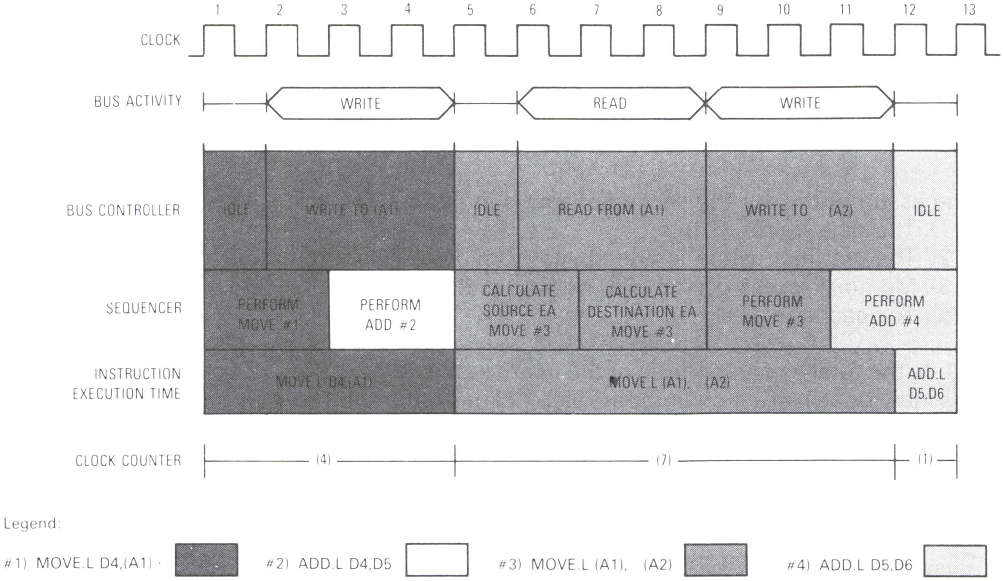

1
2
3
4
5
6
7
8
9
10
11
12
13
CLOCK
BUS ACTIVITY
WRITE
READ
WRITE
BUS CONTROLLER
IDLE
WRITE TO (A1)
IDLE
READ FROM (A1)
WRITE TO (A2)
IDLE
SEQUENCER
PERFORM MOVE #1
PERFORM ADD #2
CALCULATE SOURCE EA MOVE #3
CALCULATE DESTINATION EA MOVE #3
PERFORM MOVE #3
PERFORM ADD #4
INSTRUCTION EXECUTION TIME
MOVE.L D4,(A1)
MOVE.L (A1), (A2)
ADD.L D5,D6
CLOCK COUNTER
(4)
(7)
(1)
Legend:
#1) MOVE.L D4,(A1)
#2) ADD.L D4,D5
#3) MOVE.L (A1), (A2)
#4) ADD.L D5,D6

Figure 10-5. Processor Activity for Example 3

Note that once the instructions are in the cache, the original location in external memory is no longer a factor in timing.

Figure 10-5 illustrates the benefits of the instruction cache. The total number of clock cycles is reduced from 16 to 12 clocks. Since the instructions are resident in the cache, the instruction prefetch activity does not require the bus controller to perform external bus cycles. Prefetch occurs with no delay, and subsequently, the bus controller is idle more often.

Such idle clock cycles are useful in MC68020 systems that require wait states when accessing external memory. This is illustrated by the fourth example in Figure 10-6 with the following assumptions:

1) The data bus is 32 bits,
2) The cache is enabled and instructions are in the cache, and
3) Memory access occurs with one wait state.

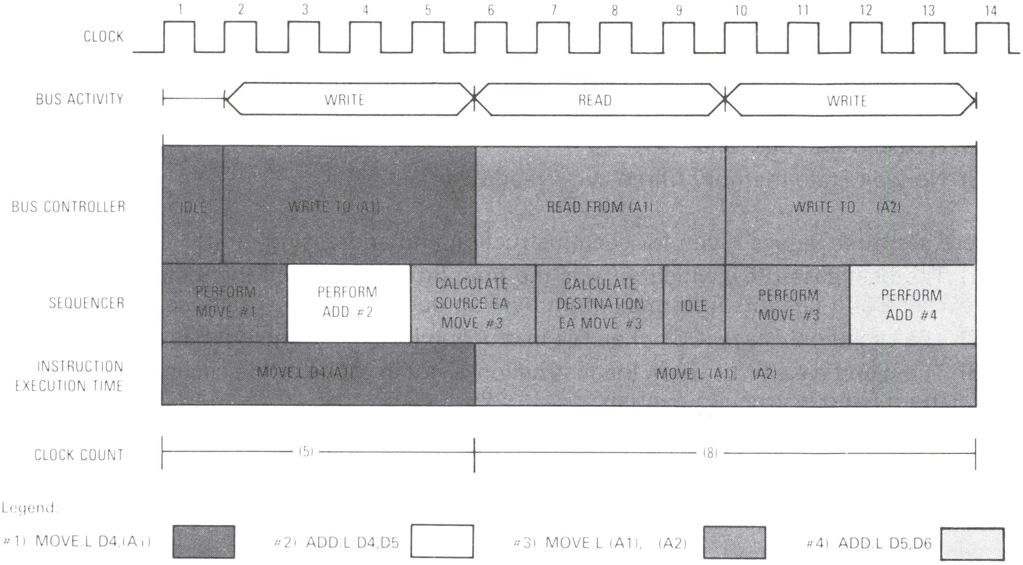

CLOCK
1
2
3
4
5
6
7
8
9
10
11
12
13
14
BUS ACTIVITY
WRITE
READ
WRITE
BUS CONTROLLER
IDLE
WRITE TO (A1)
READ FROM (A1)
WRITE TO (A2)
SEQUENCER
PERFORM MOVE #1
PERFORM ADD #2
CALCULATE SOURCE EA MOVE #3
CALCULATE DESTINATION EA MOVE #3
IDLE
PERFORM MOVE #3
PERFORM ADD #4
INSTRUCTION EXECUTION TIME
MOVE.L D4,(A1)+
MOVE.L (A1),-(A2)
CLOCK COUNT
(5)
(8)
Legend:
#1) MOVE.L D4,(A1)+
#2) ADD.L D4,D5
#3) MOVE.L (A1),-(A2)
#4) ADD.L D5,D6

Figure 10-6. Processor Activity for Example 4

Figure 10-6 shows the same instruction stream executing with four clocks for every read and write. The idle bus cycles coincide with the wait states of the memory access, so the total execution time is only 13 clocks.

These examples demonstrate the complexity of instruction timing calculation for the MC68020. It is impossible to anticipate individual instruction timing as an absolute number of clock cycles due to the dependency of overlap on the instruction sequence and alignment as well as the number of wait states in memory. This can be seen by comparing individual and composite time for Figures 10-3–10-6. These instruction timings are compared in Table 10-1, where timing varies for each instruction as the context varies.

Table 10-1. Example Instruction Stream Execution Comparison

Instruction		Example 1 (Odd Alignment)	Example 2 (Even Alignment)	Example 3 (Cache)	Example 4 (Cache with Wait States)
#1) MOVE.L	D4,(A1)+	6	4	4	5
#2) ADD.L	D4,D5	0	3	0	0
#3) MOVE.L	(A1),-(A2)	9	6	7	8
#4) ADD.L	D5,D6	1	3	1	0
Total Clock Cycles		16	16	12	13

10.2 INSTRUCTION TIMING TABLES

The instruction times below include the following assumptions about the MC68020 system:

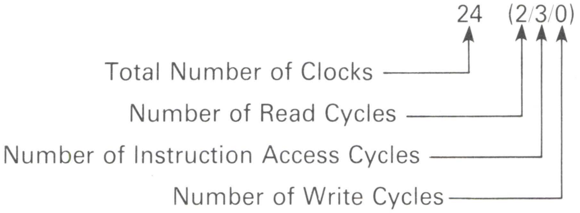
1) All operands are long-word aligned as is the stack,
2) A 32-bit data bus, and
3) No wait state memory (three-cycle read/write).

There are three values given for each instruction and addressing mode:

1) The best case (BC), which reflects the time (in clocks) when the instruction is in the cache and benefits from maximum overlap due to other instructions,
2) The cache-only case (CC) when the instruction is in the cache but has no overlap, and
3) The worst case (WC) when the instruction is not in cache or the cache is disabled and there is no instruction overlap.

The only instances for which the size of the operand has any effect are the instructions with immediate operands. Unless specified otherwise, immediate byte and word operands have identical execution times.

Within each set or column of instruction timings are four sets of numbers, three of which are enclosed in parentheses. The outer number is the total number of clocks for the instruction. The first number inside the parentheses is the number of operand read cycles performed by the instruction. The second value inside parentheses is the number of instruction accesses performed by the instruction, including all prefetches to keep the instruction pipe filled. The third value within parentheses is the number of write cycles performed by the instruction. One example from the instruction timing table is:



The total number of bus activity clocks for the above example is derived in the following way:

(2 Reads * 3 Clocks/Read) + (3 Instruction Accesses * 3 Clocks/Access)
+ (0 Writes * 3 Clocks/Write) = 15 Clocks of Bus Activity
24 Total Clocks − 15 Clocks (Bus Activity) = 9 Internal Clocks

The example used here was taken from a worst-case "fetch effective address" time. The addressing mode was ([d_{32},B],I,d_{32}). The same addressing mode under the best-case entry is 17 (2/0/0). For the best case, there are no instruction accesses because the cache is enabled, and the sequencer does not have to go to external memory for the instruction words.

The first tables deal exclusively with fetching and calculating effective addresses and immediate operands. The tables are arranged in this manner because some instructions do not require effective address calculation or fetching. For example, the instruction CLR <ea> (found in the table under **10.2.11 Single Operand Instruction**) only needs to have a calculated EA time added to its table entry because no fetch of an operand is required. This instruction only writes to memory or a register. Some instructions use specific addressing modes which exclude timing for calculation or fetching of an operand. When these instances arise, they are footnoted to indicate which other tables are needed in the timing calculation.

Many two-word instructions (e.g., MULU.L, DIV.L, BFSET, etc.) include the fetch immediate effective address time or the calculate immediate effective address time in the execution time calculation. The timing for immediate data of word length (#<data>.W) is used for these calculations. If the instruction has a source and a destination, the source EA is used for the table lookup. If the instruction is single operand, the effective address of that operand is used.

The following example includes multiword instructions that refer to the fetch immediate effective address and calculate immediate effective address tables in **10.2 INSTRUCTION TIMING TABLES**.

		Instruction
#1)	MULU.L	(D7),D1:D2
#2)	BFCLR	$6000{0:8}
#3)	DIVS.L	#$10000,D3:D4

	CC
1. MULU.L (D7),D1:D2	
#<data>.W,Dn	2
MUL.L EA, Dn	43
2. BFCLR $6000{0:8}	
#<data>.W.,$XXX.W	5
BFCLR Mem (<5 bytes)	16
3. DIVS.L #$10000,D3:D4	
#<data>.W,#<data>.L	6
DIVS.L EA,Dn	90

Execution time = 2 + 43 + 5 + 16 + 6 + 90
= 102 clock periods

NOTE

This CC time is a maximum since the times given for the MULU.L and DIVS.L are maximums.

The MOVE instruction timing tables include all necessary timing for extension word fetch, address calculation, and operand fetch.

The instruction timing tables are used to calculate a best-case and worst-case bounds for some target instruction stream. Calculating exact timing from the timing tables is impossible because the tables cannot anticipate how the combination of factors will influence every particular sequence of instructions. This is illustrated by comparing the observed instruction timing from the prior four examples with instruction timing derived from the instruction timing tables.

Table 10-2 shows the original instruction stream and the corresponding clock timing from the appropriate timing tables for the best case (BC), cache-only case (CC), and worst case (WC).

Table 10-2. Instruction Timings from Timing Tables

Instruction		Best Case	Cache Case	Worst Case
#1) MOVE.L	D4,(A1)+	4	4	6
#2) ADD.L	D4,D5	0	2	3
#3) MOVE.L	(A1),−(A2)	6	7	9
#4) ADD.L	D5,D6	0	2	3
Total		10	15	21

Table 10-3 summarizes the observed instruction timings for the same instruction stream as executed according to the assumptions of the four examples. For each example, Table 10-3 shows which entry (BC/CC/WC) from the timing tables corresponds to the observed timing for each of the four instructions. Some of the observed instruction timings cannot be found in the timing tables and appear in Table 10-3 within parenthesis in the most appropriate column. These occur when instruction execution overlap dynamically alters what would otherwise be a BC, CC, or WC timing.

Table 10-3. Observed Instruction Timings

Instruction		Example 1			Example 2			Example 3			Example 4		
		BC	CC	WC	BC	CC	WC	BC	CC	WC	BC	CC	WC
#1) MOVE.L	D4,(A1)+			6	4				4			(5)	
#2) ADD.L	D4,D5	0					3	0			0		
#3) MOVE.L	(A1),−(A2)			9	6				7			(8)	
#4) ADD.L	D5,D6	(1)					3	(1)			0		
Total			(16)			(16)			(12)			(13)	

Comparing Table 10-2 and 10-3 demonstrates that calculation of instruction timing cannot be a simple lookup of only BC or only WC timings. Even when the assumptions are known and fixed, as in the four examples summarized in Table 10-3, the microprocessor can sometimes achieve best-case timings under worst-case assumptions.

Looking across the four examples in Table 10-3 for an individual instruction, it is difficult to predict which timing table entry is used, since the influence of instruction overlap may or may not improve the BC, WC, or CC timings. When looking at the observed instruction timings for one example, it is also difficult to determine which combination of BC/CC/WC timing is required. Just how the instruction stream will fit and run with cache enabled, how instructions are positioned in memory, and the degree of instruction overlap are factors that are impossible to be accounted for in all combinations of the timing tables.

Although the timing tables cannot accurately predict the instruction timing that would be observed when executing an instruction stream on the MC68020, the tables can be used to calculate best-case and worst-case bounds for instruction timing. Absolute instruction timing must be measured by using the microprocessor itself to execute the target instruction stream.

10.2.1 Fetch Effective Address

The fetch effective address table indicates the number of clock periods needed for the processor to calculate and fetch the specified effective address. The total number of clock cycles is outside the parentheses; the number of read, pefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Address Mode	Best Case	Cache Case	Worst Case
Dn	0 (0/0/0)	0 (0/0/0)	0 (0/0/0)
An	0 (0/0/0)	0 (0/0/0)	0 (0/0/0)
(An)	3 (1/0/0)	4 (1/0/0)	4 (1/0/0)
(An)+	4 (1/0/0)	4 (1/0/0)	4 (1/0/0)
−(An)	3 (1/0/0)	5 (1/0/0)	5 (1/0/0)
(d_{16},An) of (d_{16},PC)	3 (1/0/0)	5 (1/0/0)	6 (1/1/0)
(xxx).W	3 (1/0/0)	4 (1/0/0)	6 (1/1/0)
(xxx).L	3 (1/0/0)	4 (1/0/0)	7 (1/1/0)
#<data>.B	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
#<data>.W	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
#<data>.L	0 (0/0/0)	4 (0/0/0)	5 (0/1/0)
(d_8,An,Xn) or (d_8,PC,Xn)	4 (1/0/0)	7 (1/0/0)	8 (1/1/0)
(d_{16},An,Xn) or (d_{16},PC,Xn)	4 (1/0/0)	7 (1/0/0)	9 (1/1/0)
(B)	4 (1/0/0)	7 (1/0/0)	9 (1/1/0)
(d_{16},B)	6 (1/0/0)	9 (1/0/0)	12 (1/1/0)
(d_{32},B)	10 (1/0/0)	13 (1/0/0)	16 (1/2/0)
([B],I)	9 (2/0/0)	12 (2/0/0)	13 (2/1/0)
([B],I,d_{16})	11 (2/0/0)	14 (2/0/0)	16 (2/1/0)
([B],I,d_{32})	11 (2/0/0)	14 (2/0/0)	17 (2/2/0)
([d_{16},B],I)	11 (2/0/0)	14 (2/0/0)	16 (2/1/0)
([d_{16},B],I,d_{16})	13 (2/0/0)	16 (2/0/0)	19 (2/2/0)
(d_{16},B],I,d_{32})	13 (2/0/0)	16 (2/0/0)	20 (2/2/0)
([d_{32},B],I)	15 (2/0/0)	18 (2/0/0)	20 (2/2/0)
([d_{32},B],I,d_{16})	17 (2/0/0)	20 (2/0/0)	22 (2/2/0)
([d_{32},B],I,d_{32})	17 (2/0/0)	20 (2/0/0)	24 (2/3/0)

B = Base address; 0, An, PC, Xn, An + Xn. Form does not affect timing.
I = Index; 0, Xn

NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn does not affect timing.

10.2.2 Fetch Immediate Effective Address

The fetch immediate effective address table indicates the number of clock periods needed for the processor to fetch the immediate source operand and calculate and fetch the specified destination operand. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Address Mode	Best Case	Cache Case	Worst Case
#<data>.W,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
#<data>.L,Dn	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
#<data>.W,(An)	3 (1/0/0)	4 (1/0/0)	4 (1/1/0)
#<data>.L,(An)	3 (1/0/0)	4 (1/0/0)	7 (1/1/0)
#<data>.W,(An)+	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
#<data>.L,(An)+	5 (1/0/0)	8 (1/0/0)	9 (1/1/0)
#<data>.W, –(An)	3 (1/0/0)	5 (1/0/0)	6 (1/1/0)
#<data>.L, –(An)	4 (1/0/0)	7 (1/0/0)	8 (1/1/0)
#<data>.W,(bd,An)	3 (1/0/0)	5 (1/0/0)	7 (1/1/0)
#<data>.L,(bd,An)	4 (1/0/0)	7 (1/0/0)	10 (1/2/0)
#<data>.W,xxx.W	3 (1/0/0)	5 (1/0/0)	7 (1/1/0)
#<data>.L,xxx.W	4 (1/0/0)	7 (1/0/0)	10 (1/2/0)
#<data>.W,xxx.L	3 (1/0/0)	6 (1/0/0)	10 (1/2/0)
#<data>.L,xxx.L	4 (1/0/0)	8 (1/0/0)	12 (1/2/0)
#<data>.W,#<data>.B,W	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
#<data>.W,#<data>.B,W	1 (0/0/0)	6 (0/0/0)	8 (0/2/0)
#<data>.W,#<data>.L	0 (0/0/0)	6 (0/0/0)	8 (0/2/0)
#<data>.L,#<data>.L	1 (0/0/0)	8 (0/0/0)	10 (0/2/0)
#<data>.W,(d_8,An,Xn) or (d_8,PC,Xn)	4 (1/0/0)	9 (1/0/0)	11 (1/2/0)
#<data>.L,(d_8,An,Xn) or (d_8,PC,Xn)	5 (1/0/0)	11 (1/0/0)	13 (1/2/0)
#<data>.W,(d_{16},An,Xn) or d_{16},PC,Xn)	4 (1/0/0)	9 (1/0/0)	12 (1/2/0)
#<data>.L,(d_{16},An,Xn) or d_{16},PC,Xn)	5 (1/0/0)	11 (1/0/0)	15 (1/2/0)
#<data>.W,(B)	4 (1/0/0)	9 (1/0/0)	12 (1/1/0)
#<data>.L,(B)	5 (1/0/0)	11 (1/0/0)	14 (1/2/0)
#<data>.W,(bd,PC)	10 (1/0/0)	15 (1/0/0)	19 (1/3/0)
#<data>.L,(bd,PC)	11 (1/0/0)	17 (1/0/0)	21 (1/3/0)
#<data>.W,(d_{16},B)	6 (1/0/0)	11 (1/0/0)	15 (1/2/0)
#<data>.L,(d_{16},B0	7 (1/0/0)	13 (1/0/0)	17 (1/2/0)
#<data>.W,(d_{32},B)	10 (1/0/0)	15 (1/0/0)	19 (1/3/0)
#<data>.L,(d_{32},B)	11 (1/0/0)	17 (1/0/0)	21 (1/3/0)
#<data>.W,([B],I)	9 (2/0/0)	14 (2/0/0)	16 (2/2/0)
#<data>.L,([B],I)	10 (2/0/0)	16 (2/0/0)	18 (2/2/0)
#<data>.W,([B],I,d_{16})	11 (2/0/0)	16 (2/0/0)	19 (2/2/0)

— Continued —

Address Mode	Best Case	Cache Case	Worst Case
#<data>.L,([B],I,d_{16})	12 (2/0/0)	18 (2/0/0)	21 (2/2/0)
#<data>.W,([B],I,d_{32})	11 (2/0/0)	16 (2/0/0)	20 (2/2/0)
#<data>.L,([d_{16},B],I,d_{32})	12 (2/0/0)	18 (2/0/0)	22 (2/3/0)
#<data>.W,([d_{16},B],I)	11 (2/0/0)	16 (2/0/0)	19 (2/2/0)
#<data>.L,([d_{16},B],I)	12 (2/0/0)	18 (2/0/0)	21 (2/2/0)
#<data>.W,([d_{16},B],I,d_{16})	13 (2/0/0)	18 (2/0/0)	22 (2/2/0)
#<data>.L,([d_{16},B],I,d_{16})	14 (2/0/0)	20 (2/0/0)	24 (2/3/0)
#<data>.W,([d_{32},B],I)	15 (2/0/0)	20 (2/0/0)	23 (2/3/0)
#<data>.L,([d_{32},B],I)	16 (2/0/0)	22 (2/0/0)	25 (2/3/0)
#<data>.W,([d_{32},B],I,d_{16})	17 (2/0/0)	22 (2/0/0)	25 (2/3/0)
#<data>.L,([d_{32},B],I,d_{16})	18 (2/0/0)	24 (2/0/0)	27 (2/3/0)
#<data>.W,([d_{32},b],I,d_{32})	17 (2/0/0)	22 (2/0/0)	27 (2/3/0)
#<data>.L,([d_{32},b],I,d_{32})	18 (2/0/0)	24 (2/0/0)	29 (2/4/0)

B = Base address; 0, An, PC, Xn, An + Xn. Form does not affect timing.
I = Index, 0,Xn

NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn does not affect timing.

10.2.3 Calculate Effective Address

The calculate immediate effective address table indicates the number of clock periods needed for the processor to calculate the specified effective address. Fetch time is only included for the first level of indirection on memory indirect addressing modes. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Address Mode	Best Case	Cache Case	Worst Case
Dn	0 (0/0/0)	0 (0/0/0)	0 (0/0/0)
An	0 (0/0/0)	0 (0/0/0)	0 (0/0/0)
(An)	2 (0/0/0)	2 (0/0/0)	2 (0/0/0)
(An) +	2 (0/0/0)	2 (0/0/0)	2 (0/0/0)
− (An)	2 (0/0/0)	2 (0/0/0)	2 (0/0/0)
(d_{16},An) or (d_{16},PC)	2 (0/0/0)	2 (0/0/0)	3 (0/1/0)
<data>.W	2 (0/0/0)	2 (0/0/0)	3 (0/1/0)
<data>.L	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
(d_8,An,Xn) or (d_8,PC,Xn)	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
(d_{16},An,Xn) or (d_{16},PC,Xn)	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
(B)	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
(d_{15},B)	5 (0/0/0)	8 (0/0/0)	10 (0/1/0)
(d_{32},B)	9 (0/0/0)	12 (0/0/0)	15 (0/2/0)
([B],I)	8 (1/0/0)	11 (1/0/0)	12 (1/1/0)
([B],I,d_{16})	10 (1/0/0)	13 (1/0/0)	15 (1/1/0)
([B],I,D_{32})	10 (1/0/0)	13 (1/0/0)	16 (1/2/0)
([d_{16},B],I)	10 (1/0/0)	13 (1/0/0)	15 (1/1/0)
([d_{16},B],I,d_{16})	12 (1/0/0)	15 (1/0/0)	18 (1/2/0)
([d_{16},B],I,d_{32})	12 (1/0/0)	15 (1/0/0)	19 (1/2/0)
([d_{32},B],I)	14 (1/0/0)	17 (1/0/0)	19 (1/2/0)
([d_{32},B],I,d_{16})	16 (1/0/0)	19 (1/0/0)	21 (1/2/0)
([d_{32},B],I,d_{32})	16 (1/0/0)	19 (1/0/0)	24 (1/3/0)

B = Base address; 0, An, PC, Xn, An + Xn. Form does not affect timing.
I = Index; 0, Xn

NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn does not affect timing.

10.2.4 Calculate Immediate Effective Address

The calculate immediate effective address table indicates the number of clock periods needed for the processor to fetch the immediate source operand and calculate the specified destination effective address. Fetch time is only included for the first level of indirection on memory indirect addressing modes. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Address Mode	Best Case	Cache Case	Worst Case
#<data>.W,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
#<data>.L,Dn	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
#<data>.W,(An)	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
#<data>.L,(An)	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
#<data>.W,(An)+	2 (0/0/0)	4 (0/0/0)	5 (0/1/0)
#<data>.L,(An)+	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
#<data>.W,(bd,An)	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
#<data>.L,(bd,An)	3 (0/0/0)	6 (0/0/0)	8 (0/2/0)
#<data>.W,xxx.W	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
#<data>.L,xxx.W	3 (0/0/0)	6 (0/0/0)	8 (0/2/0)
#<data>.W,xxx.L	2 (0/0/0)	4 (0/0/0)	6 (0/2/0)
#<data>.L,xxx.L	3 (0/0/0)	8 (0/0/0)	10 (0/2/0)
#<data>.W,(d_8,An,Xn) or (d_8,PC,Xn)	0 (0/0/0)	6 (0/0/0)	8 (0/2/0)
#<data>.L,(d_8,An,Xn) or (d_8,PC,Xn)	2 (0/0/0)	8 (0/0/0)	10 (0/2/0)
#<data>.W,(d_{16},An,Xn) or d_{16},PC,Xn)	3 (0/0/0)	8 (0/0/0)	10 (0/2/0)
#<data>.L,(d_{16},An,Xn) or d_{16},PC,Xn)	4 (0/0/0)	10 (0/0/0)	12 (0/2/0)
#<data>.W,(B)	3 (0/0/0)	8 (0/0/0)	10 (0/1/0)
#<data>.L,(B)	4 (0/0/0)	10 (0/0/0)	12 (0/2/0)
#<data>.W,(bd,PC)	9 (0/0/0)	14 (0/0/0)	18 (0/3/0)
#<data>.L,(bd,PC)	10 (0/0/0)	16 (0/0/0)	20 (0/3/0)
#<data>.W,(d_{16},B)	5 (0/0/0)	10 (0/0/0)	13 (0/2/0)
#<data>.L,(d_{16},B)	6 (0/0/0)	12 (0/0/0)	15 (0/2/0)
#<data>.W,(d_{32},B)	9 (0/0/0)	14 (0/0/0)	18 (0/2/0)
#<data>.L,(d_{32},B)	10 (0/0/0)	16 (0/0/0)	20 (0/3/0)
#<data>.W,([B],I)	8 (1/0/0)	13 (1/0/0)	15 (1/2/0)
#<data>.L,([B],I)	9 (1/0/0)	15 (1/0/0)	17 (1/2/0)
#<data>.W,([B],I,d_{16})	10 (1/0/0)	15 (1/0/0)	18 (1/2/0)
#<data>.L,([B],I,d_{16})	11 (1/0/0)	17 (1/0/0)	20 (1/2/0)
#<data>.W,([B],I,d_{32})	10 (1/0/0)	15 (1/0/0)	19 (1/2/0)
#<data>.L,([d_{16},B],I,d_{32})	11 (1/0/0)	17 (1/0/0)	21 (1/3/0)
#<data>.W,([d_{16},B],I)	10 (1/0/0)	15 (1/0/0)	18 (1/2/0)
#<data>.L,([d_{16},B],I)	11 (1/0/0)	17 (1/0/0)	20 (1/2/0)

— Continued —

Address Mode	Best Case	Cache Case	Worst Case
#<data>.W,([d$_{16}$,B],I,d$_{16}$)	12 (1/0/0)	17 (1/0/0)	21 (1/2/0)
#<data>.L,([d$_{16}$,B],I,d$_{16}$)	13 (1/0/0)	19 (1/0/0)	23 (1/3/0)
#<data>.([d$_{16}$,B],I,d$_{32}$)	12 (1/0/0)	17 (1/0/0)	22 (1/3/0)
#<data>.([d$_{16}$,B],I,d$_{32}$)	13 (1/0/0)	19 (1/0/0)	24 (1/3/0)
#<data>.W,([d$_{32}$,B],I)	14 (1/0/0)	19 (1/0/0)	22 (1/3/0)
#<data>.L,([d$_{32}$,B],I)	15 (1/0/0)	21 (1/0/0)	24 (1/3/0)
#<data>.W,([d$_{32}$,B],I,d$_{16}$)	16 (1/0/0)	21 (1/0/0)	24 (1/3/0)
#<data>.L,([d$_{32}$,B],I,d$_{16}$)	17 (1/0/0)	23 (1/0/0)	26 (1/3/0)
#<data>.W,([d$_{32}$,b],I,d$_{32}$)	16 (1/0/0)	21 (1/0/0)	24 (1/3/0)
#<data>.L,([d$_{32}$,b],I,d$_{32}$)	17 (1/0/0)	23 (1/0/0)	29 (1/4/0)

B = Base address; 0, An, PC, Xn, An + Xn. Form does not affect timing.
I = Index; 0,Xn

NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn does not affect timing.

10.2.5 Jump Effective Address

The jump effective address table indicates the number of clock periods needed for the processor to calculate the specified effective address. Fetch time is only included for the first level of indirection on memory indirect addressing modes. The total number of clock cycles is outside the parentheses; the number or read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Address Mode	Best Case	Cache Case	Worst Case
(An)	0 (0/0/0)	2 (0/0/0)	2 (0/0/0)
(d_{16},An)	1 (0/0/0)	4 (0/0/0)	4 (0/0/0)
(xxx).W	0 (0/0/0)	2 (0/0/0)	2 (0/0/0)
(xxx).L	0 (0/0/0)	2 (0/0/0)	2 (0/0/0)
(d_8,An,Xn) or (d_8,PC,Xn)	3 (0/0/0)	6 (0/0/0)	6 (0/0/0)
(d_{16}An,Xn) or (d_{16},PC,Xn)	3 (0/0/0)	6 (0/0/0)	6 (0/0/0)
(B)	3 (0/0/0)	6 (0/0/0)	6 (0/0/0)
(B,d_{16})	5 (0/0/0)	8 (0/0/0)	8 (0/1/0)
(b,d_{32})	9 (0/0/0)	12 (0/0/0)	12 (0/1/0)
([B],I)	8 (1/0/0)	11 (1/0/0)	11 (1/1/0)
([B],I,d_{16})	10 (1/0/0)	13 (1/0/0)	14 (1/1/0)
([B],I,d_{32})	10 (1/0/0)	13 (1/0/0)	14 (1/1/0)
(d[$_{16}$,B],I)	10 (1/0/0)	13 (1/0/0)	14 (1/1/0)
([d_{16},B],I,d_{16})	12 (1/0/0)	15 (1/0/0)	17 (1/1/0)
([d_{16},B],I,d_{32})	12 (1/0/0)	15 (1/0/0)	17 (1/1/0)
([d_{32},B],I)	14 (1/0/0)	17 (1/0/0)	19 (1/2/0)
([d_{32},B],I,d_{16})	16 (1/0/0)	19 (1/0/0)	21 (1/2/0)
([d_{32},B],I,d_{32})	16 (1/0/0)	19 (1/0/0)	23 (1/2/0)

B = Base address; 0, An, PC, Xn, An + Xn, PC + Xn. Form does not affect timing.
I = Index; 0, Xn

NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn does not affect timing.

10.2.6 MOVE Instruction

The MOVE instruction timing table indicates the number of clock periods needed for the processor to fetch, calculate, and perform the MOVE or MOVEA with the specified source and destination effective addresses, including both levels of indirection on memory indirect addressing modes. No additional tables are needed to calculate the total effective execution time for the MOVE or MOVEA instruction. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

BEST CASE

Source Address Mode	**Destination**							
	An	**Dn**	**(An)**	**(An)+**	**−(An)**	**(d$_{16}$,An)**	**(xxx).W**	**(xxx).L**
Rn	0 (0/0/0)	0 (0/0/0)	3 (0/0/1)	4 (0/0/1)	3 (0/0/1)	3 (0/0/1)	3 (0/0/1)	5 (0/0/1)
#<data>.B,W	0 (0/0/0)	0 (0/0/0)	3 (0/0/1)	4 (0/0/1)	3 (0/0/1)	3 (0/0/1)	3 (0/0/1)	5 (0/0/1)
#<data>.L	0 (0/0/0)	0 (0/0/0)	3 (0/0/1)	4 (0/0/1)	3 (0/0/1)	3 (0/0/1)	3 (0/0/1)	5 (0/0/1)
(An)	3 (1/0/0)	3 (1/0/0)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	8 (1/0/1)
(An)+	4 (1/0/0)	4 (1/0/0)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	9 (1/0/1)
−(An)	3 (1/0/0)	3 (1/0/0)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	8 (1/0/1)
(d$_{16}$,An) or (d$_{16}$,PC)	3 (1/0/0)	3 (1/0/0)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	8 (1/0/1)
(xxx).W	3 (1/0/0)	3 (1/0/0)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	8 (1/0/1)
(xxx).L	3 (1/0/0)	3 (1/0/0)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	6 (1/0/1)	8 (1/0/1)
(d$_8$,An,Xn) or d$_8$,PC,Xn)	4 (1/0/0)	4 (1/0/0)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	9 (1/0/1)
(d$_{16}$,An,Xn) or (d$_{16}$,PC,Xn)	4 (1/0/0)	4 (1/0/0)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	9 (1/0/1)
(B)	4 (1/0/0)	4 (1/0/0)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	9 (1/0/1)
(d$_{16}$,B)	6 (1/0/0)	6 (1/0/0)	9 (1/0/1)	9 (1/0/1)	9 (1/0/1)	9 (1/0/1)	9 (1/0/1)	11 (1/0/1)
(d$_{32}$,B)	10 (1/0/0)	10 (1/0/0)	13 (1/0/1)	13 (1/0/1)	13 (1/0/1)	13 (1/0/1)	13 (1/0/1)	15 (1/0/1)
([B],I)	9 (2/0/0)	9 (2/0/0)	12 (2/0/1)	12 (2/0/1)	12 (2/0/1)	12 (2/0/1)	12 (2/0/1)	14 (2/0/1)
([B],I,d$_{16}$)	11 (2/0/0)	11 (2/0/0)	14 (2/0/1)	14 (2/0/1)	14 (2/0/1)	14 (2/0/1)	14 (2/0/1)	16 (2/0/1)
([B],I,D$_{32}$)	11 (2/0/0)	11 (2/0/0)	14 (2/0/1)	14 (2/0/1)	14 (2/0/1)	14 (2/0/1)	14 (2/0/1)	16 (2/0/1)
([d$_{16}$,B],I)	11 (2/0/0)	11 (2/0/0)	14 (2/0/1)	14 (2/0/1)	14 (2/0/1)	14 (2/0/1)	14 (2/0/1)	16 (2/0/1)
([d$_{16}$,B],I,d$_{16}$)	13 (2/0/0)	13 (2/0/0)	16 (2/0/1)	16 (2/0/1)	16 (2/0/1)	16 (2/0/1)	16 (2/0/1)	18 (2/0/1)
([d$_{16}$,B],d$_{32}$)	13 (2/0/0)	13 (2/0/0)	16 (2/0/1)	16 (2/0/1)	16 (2/0/1)	16 (2/0/1)	16 (2/0/1)	18 (2/0/1)
([d$_{32}$,B],I)	15 (2/0/0)	15 (2/0/0)	18 (2/0/1)	18 (2/0/1)	18 (2/0/1)	18 (2/0/1)	18 (2/0/1)	20 (2/0/1)
([d$_{32}$,B],I,d$_{15}$)	17 (2/0/0)	17 (2/0/0)	20 (2/0/1)	20 (2/0/1)	20 (2/0/1)	20 (2/0/1)	20 (2/0/1)	22 (2/0/1)
([d$_{32}$,B],I,d$_{32}$)	17 (2/0/0)	17 (2/0/0)	20 (2/0/1)	20 (2/0/1)	20 (2/0/1)	20 (2/0/1)	20 (2/0/1)	22 (2/0/1)

BEST CASE (Continued)

Source Address Mode	Destination							
	(d$_8$,An,Xn)	**(d$_{16}$,An,Xn)**	**(B)**	**(d$_{16}$,B)**	**(d$_{32}$,B)**	**([B],I)**	**([B],I,d$_{16}$)**	**([B],I,D$_{32}$)**
Rn	4 (0/0/1)	6 (0/0/1)	5 (0/0/1)	7 (0/0/1)	11 (0/0/1)	9 (1/0/1)	11 (1/0/1)	12 (1/0/1)
#<data>.B,W	4 (0/0/1)	6 (0/0/1)	5 (0/0/1)	7 (0/0/1)	11 (0/0/1)	9 (1/0/1)	11 (1/0/1)	12 (1/0/1)
#<data>.L	4 (0/0/1)	6 (0/0/1)	5 (0/0/1)	7 (0/0/1)	11 (0/0/1)	9 (1/0/1)	11 (1/0/1)	12 (1/0/1)
(An)	8 (1/0/1)	10 (1/0/1)	9 (1/0/1)	11 (1/0/1)	15 (1/0/1)	13 (2/0/1)	15 (2/0/1)	16 (2/0/1)
(An)+	9 (1/0/1)	11 (1/0/1)	10 (1/0/1)	12 (1/0/1)	16 (1/0/1)	14 (2/0/1)	16 (2/0/1)	17 (2/0/1)
−(An)	8 (1/0/1)	10 (1/0/1)	9 (1/0/1)	11 (1/0/1)	15 (1/0/1)	13 (2/0/1)	15 (2/0/1)	16 (2/0/1)
(d$_{16}$,An) or (d$_{16}$,PC)	8 (1/0/1)	10 (1/0/1)	9 (1/0/1)	11 (1/0/1)	15 (1/0/1)	13 (2/0/1)	15 (2/0/1)	16 (2/0/1)
(xxx).W	8 (1/0/1)	10 (1/0/1)	9 (1/0/1)	11 (1/0/1)	15 (1/0/1)	13 (2/0/1)	15 (2/0/1)	16 (2/0/1)
(xxx).L	8 (1/0/1)	10 (1/0/1)	9 (1/0/1)	11 (1/0/1)	15 (1/0/1)	13 (2/0/1)	15 (2/0/1)	16 (2/0/1)
(d$_8$,An,Xn) or (d$_8$,PC,Xn)	9 (1/0/1)	10 (1/0/1)	10 (1/0/1)	12 (1/0/1)	16 (1/0/1)	14 (2/0/1)	16 (2/0/1)	17 (2/0/1)
(d$_{16}$,An,Xn) or (d$_{16}$,PC,Xn)	9 (1/0/1)	11 (1/0/1)	10 (1/0/1)	12 (1/0/1)	16 (1/0/1)	14 (2/0/1)	16 (2/0/1)	17 (2/0/1)
(B)	9 (1/0/1)	11 (1/0/1)	10 (1/0/1)	12 (1/0/1)	16 (1/0/1)	14 (2/0/1)	16 (2/0/1)	17 (2/0/1)
(d$_{16}$,B)	11 (1/0/1)	13 (1/0/1)	12 (1/0/1)	14 (1/0/1)	18 (1/0/1)	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)
(d$_{32}$,B)	15 (1/0/1)	17 (1/0/1)	18 (1/0/1)	18 (1/0/1)	22 (1/0/1)	20 (2/0/1)	22 (2/0/1)	23 (2/0/1)
([B],I)	14 (2/0/1)	16 (2/0/1)	17 (2/0/1)	17 (2/0/1)	21 (2/0/1)	19 (3/0/1)	21 (3/0/1)	22 (3/0/1)
([B],I,D$_{16}$)	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)	19 (2/0/1)	23 (2/0/1)	21 (3/0/1)	23 (3/0/1)	24 (3/0/1)
([B],I,d$_{32}$)	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)	19 (2/0/1)	23 (2/0/1)	21 (3/0/1)	23 (3/0/1)	24 (3/0/1)
([d$_{16}$,B],I)	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)	19 (2/0/1)	23 (2/0/1)	21 (3/0/1)	23 (3/0/1)	24 (3/0/1)
([d$_{16}$,B],I,d$_{16}$)	18 (2/0/1)	20 (2/0/1)	21 (2/0/1)	21 (2/0/1)	25 (2/0/1)	23 (3/0/1)	25 (3/0/1)	26 (3/0/1)
([d$_{16}$,B],I,d$_{32}$)	18 (2/0/1)	20 (2/0/1)	21 (2/0/1)	21 (2/0/1)	25 (2/0/1)	23 (3/0/1)	25 (3/0/1)	26 (3/0/1)
([d$_{32}$,B],I)	20 (2/0/1)	22 (2/0/1)	23 (2/0/1)	23 (2/0/1)	27 (2/0/1)	25 (3/0/1)	27 (3/0/1)	28 (3/0/1)
([d$_{32}$,B],I,d$_{16}$)	22 (2/0/1)	24 (2/0/1)	25 (2/0/1)	25 (2/0/1)	29 (2/0/1)	27 (3/0/1)	29 (3/0/1)	30 (3/0/1)
([d$_{32}$,B],I,d$_{32}$)	22 (2/0/1)	24 (2/0/1)	25 (2/0/1)	25 (2/0/1)	29 (2/0/1)	27 (3/0/1)	29 (3/0/1)	30 (3/0/1)

BEST CASE (Concluded)

Source Address Mode	**Destination**					
	([d_{16},B],I)	**([d_{16},B],I,d_{16})**	**([d_{16},B],I,d_{32})**	**([d_{32},B],I)**	**([d_{32},B],I,d_{16})**	**([d_{32},B],I,d_{32})**
Rn	11 (1/0/1)	13 (1/0/1)	14 (1/0/1)	15 (1/0/1)	17 (1/0/1)	18 (1/0/1)
#<data>.B,W	11 (1/0/1)	13 (1/0/1)	14 (1/0/1)	15 (1/0/1)	17 (1/0/1)	18 (1/0/1)
#<data>.L	11 (1/0/1)	13 (1/0/1)	14 (1/0/1)	15 (1/0/1)	17 (1/0/1)	18 (1/0/1)
(An)	15 (2/0/1)	17 (2/0/1)	18 (2/0/1)	19 (2/0/1)	21 (2/0/1)	22 (2/0/1)
(An)+	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)	20 (2/0/1)	22 (2/0/1)	23 (2/0/1)
−(An)	15 (2/0/1)	17 (2/0/1)	18 (2/0/1)	19 (2/0/1)	21 (2/0/1)	22 (2/0/1)
(d_{16},An) or (d_{16},PC)	15 (2/0/1)	17 (2/0/1)	18 (2/0/1)	19 (2/0/1)	21 (2/0/1)	22 (2/0/1)
(xxx).W	15 (2/0/1)	17 (2/0/1)	18 (2/0/1)	19 (2/0/1)	21 (2/0/1)	22 (2/0/1)
(xxx).L	15 (2/0/1)	17 (2/0/1)	18 (2/0/1)	19 (2/0/1)	21 (2/0/1)	22 (2/0/1)
(d_8,An,Xn) or (d_8,PC,Xn)	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)	20 (2/0/1)	22 (2/0/1)	23 (2/0/1)
(d_{16},An,Xn) or (d_{16},PC,Xn)	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)	20 (2/0/1)	22 (2/0/1)	23 (2/0/1)
(B)	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)	20 (2/0/1)	22 (2/0/1)	23 (2/0/1)
(d_{16},B)	18 (2/0/1)	20 (2/0/1)	21 (2/0/1)	22 (2/0/1)	24 (2/0/1)	25 (2/0/1)
(d_{32},B)	22 (2/0/1)	24 (2/0/1)	25 (2/0/1)	26 (2/0/1)	28 (2/0/1)	29 (2/0/1)
([B],I)	21 (3/0/1)	23 (3/0/1)	24 (3/0/1)	25 (3/0/1)	27 (3/0/1)	28 (3/0/1)
([B],I,d_{16})	23 (3/0/1)	25 (3/0/1)	26 (3/0/1)	27 (3/0/1)	29 (3/0/1)	30 (3/0/1)
([B],I,d_{32})	23 (3/0/1)	25 (3/0/1)	26 (3/0/1)	27 (3/0/1)	29 (3/0/1)	30 (3/0/1)
([d_{16},B],I)	23 (3/0/1)	25 (3/0/1)	26 (3/0/1)	27 (3/0/1)	29 (3/0/1)	30 (3/0/1)
([d_{16},B],I,d_{16})	25 (3/0/1)	27 (3/0/1)	28 (3/0/1)	29 (3/0/1)	31 (3/0/1)	32 (3/0/1)
([d_{16},B],I,d_{32})	25 (3/0/1)	27 (3/0/1)	28 (3/0/1)	29 (3/0/1)	31 (3/0/1)	32 (0/0/1)
([d_{32},B],I)	27 (3/0/1)	29 (3/0/1)	30 (3/0/1)	31 (3/0/1)	33 (3/0/1)	34 (3/0/1)
([d_{32},B],I,d_{16})	29 (3/0/1)	31 (3/0/1)	32 (3/0/1)	33 (3/0/1)	35 (3/0/1)	36 (3/0/1)
([d_{32},B],I,d_{32})	29 (3/0/1)	31 (3/0/1)	32 (3/0/1)	33 (3/0/1)	35 (3/0/1)	36 (3/0/1)

CACHE CASE

Source Address Mode	Destination							
	An	**Dn**	**(An)**	**(An) +**	**− (An)**	**(d_{16},An)**	**(xxx).W**	**(xxx).L**
Rn	2 (0/0/0)	2 (0/0/0)	4 (0/0/1)	4 (0/0/1)	5 (0/0/1)	5 (0/0/1)	4 (0/0/1)	6 (0/0/1)
#<data>.B,W	4 (0/0/0)	4 (0/0/0)	6 (0/0/1)	6 (0/0/1)	7 (0/0/1)	7 (0/0/1)	6 (0/0/1)	8 (0/0/1)
#<data>.L	6 (0/0/0)	6 (0/0/0)	8 (0/0/1)	8 (0/0/1)	9 (0/0/1)	9 (0/0/1)	8 (0/0/1)	10 (0/0/1)
(An)	6 (1/0/0)	6 (1/0/0)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	9 (1/0/1)
(An) +	6 (1/0/0)	6 (1/0/0)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	9 (1/0/1)
− (An)	7 (1/0/0)	7 (1/0/0)	8 (1/0/1)	8 (1/0/1)	8 (1/0/1)	8 (1/0/1)	8 (1/0/1)	10 (1/0/1)
(d_{16},An) or (d_{16},PC)	7 (1/0/0)	7 (1/0/0)	8 (1/0/1)	8 (1/0/1)	8 (1/0/1)	8 (1/0/1)	8 (1/0/1)	10 (1/0/1)
(xxx).W	6 (1/0/0)	6 (1/0/0)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	9 (1/0/1)
(xxx).L	6 (1/0/0)	6 (1/0/0)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	7 (1/0/1)	9 (1/0/1)
(d_8,An,Xn) or d_8,PC,Xn)	9 (1/0/0)	9 (1/0/0)	10 (1/0/1)	10 (1/0/1)	10 (1/0/1)	10 (1/0/1)	10 (1/0/1)	12 (1/0/1)
(d_{16},An,Xn) or (d_{16},PC,Xn)	9 (1/0/0)	9 (1/0/0)	10 (1/0/1)	10 (1/0/1)	10 (1/0/1)	10 (1/0/1)	10 (1/0/1)	12 (1/0/1)
(B)	9 (1/0/0)	9 (1/0/0)	10 (1/0/1)	10 (1/0/1)	10 (1/0/1)	10 (1/0/1)	10 (1/0/1)	12 (1/0/1)
(d_{16},B)	11 (1/0/0)	11 (1/0/0)	12 (1/0/1)	12 (1/0/1)	12 (1/0/1)	12 (1/0/1)	12 (1/0/1)	14 (1/0/1)
(d_{32},B)	15 (1/0/0)	15 (1/0/0)	16 (1/0/1)	16 (1/0/1)	16 (1/0/1)	16 (1/0/1)	16 (1/0/1)	18 (1/0/1)
([B],I)	14 (2/0/0)	14 (2/0/0)	15 (2/0/1)	15 (2/0/1)	15 (2/0/1)	15 (2/0/1)	15 (2/0/1)	17 (2/0/1)
([B],I,d_{16})	16 (2/0/0)	16 (2/0/0)	17 (2/0/1)	17 (2/0/1)	17 (2/0/1)	17 (2/0/1)	17 (2/0/1)	19 (2/0/1)
([B],I,D_{32})	16 (2/0/0)	16 (2/0/0)	17 (2/0/1)	17 (2/0/1)	17 (2/0/1)	17 (2/0/1)	17 (2/0/1)	19 (2/0/1)
([d_{16},B],I)	16 (2/0/0)	16 (2/0/0)	17 (2/0/1)	17 (2/0/1)	17 (2/0/1)	17 (2/0/1)	17 (2/0/1)	19 (2/0/1)
([d_{16},B],I,d_{16})	18 (2/0/0)	18 (2/0/0)	19 (2/0/1)	19 (2/0/1)	19 (2/0/1)	19 (2/0/1)	19 (2/0/1)	21 (2/0/1)
([d_{16},B],d_{32})	18 (2/0/0)	18 (2/0/0)	19 (2/0/1)	19 (2/0/1)	19 (2/0/1)	19 (2/0/1)	19 (2/0/1)	21 (2/0/1)
([d_{32},B],I)	20 (2/0/0)	20 (2/0/0)	21 (2/0/1)	21 (2/0/1)	21 (2/0/1)	21 (2/0/1)	21 (2/0/1)	23 (2/0/1)
([d_{32},B],I,d_{15})	22 (2/0/0)	22 (2/0/0)	23 (2/0/1)	23 (2/0/1)	23 (2/0/1)	23 (2/0/1)	23 (2/0/1)	25 (2/0/1)
([d_{32},B],I,d_{32})	22 (2/0/0)	22 (2/0/0)	23 (2/0/1)	23 (2/0/1)	23 (2/0/1)	23 (2/0/1)	23 (2/0/1)	25 (2/0/1)

CACHE CASE (Continued)

Source Address Mode	Destination							
	(d_8,An,Xn)	**(d_{16},An,Xn)**	**(B)**	**(d_{16},B)**	**(d_{32},B)**	**([B],I)**	**([B],I,d_{16})**	**([B],I,D_{32})**
Rn	7 (0/0/1)	9 (0/0/1)	8 (0/0/1)	10 (0/0/1)	14 (0/0/1)	12 (1/0/1)	14 (1/0/1)	15 (1/0/1)
#<data>.B,W	7 (0/0/1)	9 (0/0/1)	8 (0/0/1)	10 (0/0/1)	14 (0/0/1)	12 (1/0/1)	14 (1/0/1)	15 (1/0/1)
#<data>.L	9 (0/0/1)	11 (0/0/1)	10 (0/0/1)	12 (0/0/1)	16 (0/0/1)	14 (1/0/1)	16 (1/0/1)	17 (1/0/1)
(An)	9 (1/0/1)	11 (1/0/1)	10 (1/0/1)	12 (1/0/1)	16 (1/0/1)	14 (2/0/1)	16 (2/0/1)	17 (2/0/1)
(An)+	9 (1/0/1)	11 (1/0/1)	10 (1/0/1)	12 (1/0/1)	16 (1/0/1)	14 (2/0/1)	16 (2/0/1)	17 (2/0/1)
−(An)	10 (1/0/1)	12 (1/0/1)	11 (1/0/1)	13 (1/0/1)	17 (1/0/1)	15 (2/0/1)	17 (2/0/1)	18 (2/0/1)
(d_{16},An) or (d_{16},PC)	10 (1/0/1)	12 (2/0/1)	11 (1/0/1)	13 (1/0/1)	17 (1/0/1)	15 (2/0/1)	17 (2/0/1)	18 (2/0/1)
(xxx).W	9 (1/0/1)	11 (1/0/1)	10 (1/0/1)	12 (1/0/1)	16 (1/0/1)	14 (2/0/1)	16 (2/0/1)	17 (2/0/1)
(xxx).L	9 (1/0/1)	11 (1/0/1)	10 (1/0/1)	12 (1/0/1)	16 (1/0/1)	14 (2/0/1)	16 (2/0/1)	17 (2/0/1)
(d_8,An,Xn) or (d_8,PC,Xn)	12 (1/0/1)	14 (1/0/1)	13 (1/0/1)	15 (1/0/1)	19 (1/0/1)	17 (2/0/1)	19 (2/0/1)	20 (2/0/1)
(d_{16},An,Xn) or (d_{16},PC,Xn)	12 (1/0/1)	14 (1/0/1)	13 (1/0/1)	15 (1/0/1)	19 (1/0/1)	17 (2/0/1)	19 (2/0/1)	20 (2/0/1)
(B)	12 (1/0/1)	14 (1/0/1)	13 (1/0/1)	15 (1/0/1)	19 (1/0/1)	17 (2/0/1)	19 (2/0/1)	20 (2/0/1)
(d_{16},B)	14 (1/0/1)	16 (1/0/1)	15 (1/0/1)	17 (1/0/1)	21 (1/0/1)	19 (2/0/1)	21 (2/0/1)	22 (2/0/1)
(d_{32},B)	18 (1/0/1)	20 (1/0/1)	19 (1/0/1)	21 (1/0/1)	25 (1/0/1)	23 (2/0/1)	25 (2/0/1)	26 (2/0/1)
([B],I)	17 (2/0/1)	19 (2/0/1)	18 (2/0/1)	20 (2/0/1)	24 (2/0/1)	22 (3/0/1)	24 (3/0/1)	25 (3/0/1)
([B],I,D_{16})	19 (2/0/1)	21 (2/0/1)	20 (2/0/1)	22 (2/0/1)	26 (2/0/1)	24 (3/0/1)	26 (3/0/1)	27 (3/0/1)
([B],I,d_{32})	19 (2/0/1)	21 (2/0/1)	20 (2/0/1)	22 (2/0/1)	26 (2/0/1)	24 (3/0/1)	26 (3/0/1)	27 (3/0/1)
([d_{16},B],I)	19 (2/0/1)	21 (2/0/1)	20 (2/0/1)	22 (2/0/1)	26 (2/0/1)	24 (3/0/1)	26 (3/0/1)	27 (3/0/1)
([d_{16},B],I,d_{16})	21 (2/0/1)	23 (2/0/1)	22 (2/0/1)	24 (2/0/1)	28 (2/0/1)	26 (3/0/1)	28 (3/0/1)	29 (3/0/1)
([d_{16},B],I,d_{32})	21 (2/0/1)	23 (2/0/1)	22 (2/0/1)	24 (2/0/1)	28 (2/0/1)	26 (3/0/1)	28 (3/0/1)	29 (3/0/1)
([d_{32},B],I)	23 (2/0/1)	25 (2/0/1)	24 (2/0/1)	26 (2/0/1)	30 (2/0/1)	28 (3/0/1)	30 (3/0/1)	31 (3/0/1)
([d_{32},B],I,d_{16})	25 (2/0/1)	27 (2/0/1)	26 (2/0/1)	28 (2/0/1)	32 (2/0/1)	30 (3/0/1)	32 (3/0/1)	33 (3/0/1)
([d_{32},B],I,d_{32})	25 (2/0/1)	27 (2/0/1)	26 (2/0/1)	28 (2/0/1)	32 (2/0/1)	30 (3/0/1)	32 (3/0/1)	33 (3/0/1)

CACHE CASE (Concluded)

Source Address Mode	Destination					
	$([d_{16},B],I)$	$([d_{16},B],I,d_{16})$	$([d_{16},B],I,d_{32})$	$([d_{32},B],I)$	$([d_{32},B],I,d_{16})$	$([d_{32},B],I,d_{32})$
Rn	14 (1/0/1)	16 (1/0/1)	17 (1/0/1)	18 (1/0/1)	20 (1/0/1)	21 (1/0/1)
#<data>.B,W	14 (1/0/1)	16 (1/0/1)	17 (1/0/1)	18 (1/0/1)	20 (1/0/1)	21 (1/0/1)
#<data>.L	16 (1/0/1)	18 (1/0/1)	19 (1/0/1)	20 (1/0/1)	22 (1/0/1)	23 (1/0/1)
(An)	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)	20 (2/0/1)	22 (2/0/1)	23 (2/0/1)
(An)+	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)	20 (2/0/1)	22 (2/0/1)	23 (2/0/1)
−(An)	17 (2/0/1)	19 (2/0/1)	20 (2/0/1)	21 (2/0/1)	23 (2/0/1)	24 (2/0/1)
(d_{16},An) or (d_{16},PC)	17 (2/0/1)	19 (2/0/1)	20 (2/0/1)	21 (2/0/1)	23 (2/0/1)	24 (2/0/1)
(xxx).W	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)	20 (2/0/1)	22 (2/0/1)	23 (2/0/1)
(xxx).L	16 (2/0/1)	18 (2/0/1)	19 (2/0/1)	20 (2/0/1)	22 (2/0/1)	23 (2/0/1)
(d_8,An,Xn) or (d_8,PC,Xn)	19 (2/0/1)	21 (2/0/1)	22 (2/0/1)	23 (2/0/1)	25 (2/0/1)	26 (2/0/1)
(d_{16},An,Xn) or (d_{16},PC,Xn)	19 (2/0/1)	21 (2/0/1)	22 (2/0/1)	23 (2/0/1)	25 (2/0/1)	26 (2/0/1)
(B)	19 (2/0/1)	21 (2/0/1)	22 (2/0/1)	23 (2/0/1)	25 (2/0/1)	26 (2/0/1)
(d_{16},B)	21 (2/0/1)	23 (2/0/1)	24 (2/0/1)	25 (2/0/1)	27 (2/0/1)	28 (2/0/1)
(d_{32},B)	25 (2/0/1)	27 (2/0/1)	28 (2/0/1)	29 (2/0/1)	31 (2/0/1)	32 (2/0/1)
([B],I)	24 (3/0/1)	26 (3/0/1)	27 (3/0/1)	28 (3/0/1)	30 (3/0/1)	31 (3/0/1)
$([B],I,d_{16})$	26 (3/0/1)	28 (3/0/1)	29 (3/0/1)	30 (3/0/1)	32 (3/0/1)	33 (3/0/1)
$([B],I,d_{32})$	26 (3/0/1)	28 (3/0/1)	29 (3/0/1)	30 (3/0/1)	32 (3/0/1)	33 (3/0/1)
$([d_{16},B],I)$	26 (3/0/1)	28 (3/0/1)	29 (3/0/1)	30 (3/0/1)	32 (3/0/1)	33 (3/0/1)
$([d_{16},B],I,d_{16})$	28 (3/0/1)	30 (3/0/1)	31 (3/0/1)	32 (3/0/1)	34 (3/0/1)	35 (3/0/1)
$([d_{16},B],I,d_{32})$	28 (3/0/1)	30 (3/0/1)	31 (3/0/1)	32 (3/0/1)	34 (3/0/1)	35 (3/0/1)
$([d_{32},B],I)$	30 (3/0/1)	32 (3/0/1)	33 (3/0/1)	34 (3/0/1)	36 (3/0/1)	37 (3/0/1)
$([d_{32},B],I,d_{16})$	32 (3/0/1)	34 (3/0/1)	35 (3/0/1)	36 (3/0/1)	38 (3/0/1)	39 (3/0/1)
$([d_{32},B],I,d_{32})$	32 (3/0/1)	34 (3/0/1)	35 (3/0/1)	36 (3/0/1)	38 (3/0/1)	39 (3/0/1)

WORST CASE

Source Address Mode	Destination							
	An	Dn	(An)	(An)+	−(An)	(d_{16},An)	(xxx).W	(xxx).L
Rn	3 (0/1/0)	3 (0/1/0)	5 (0/1/0)	5 (0/1/1)	6 (0/1/1)	7 (0/1/1)	7 (0/1/1)	9 (0/2/1)
#<data>.B,W	3 (0/1/0)	3 (0/1/0)	5 (0/1/0)	8 (0/1/1)	6 (0/1/1)	7 (0/1/1)	7 (0/1/1)	9 (0/2/1)
#<data>.L	5 (0/1/0)	5 (0/1/0)	7 (0/0/1)	7 (0/1/1)	8 (0/1/1)	9 (0/1/1)	9 (0/1/1)	11 (0/2/1)
(An)	7 (1/1/0)	7 (1/1/0)	9 (1/1/1)	9 (1/1/1)	9 (1/1/1)	11 (1/1/1)	11 (1/1/1)	13 (1/2/1)
(An)+	7 (1/1/0)	7 (1/1/0)	9 (1/1/1)	9 (1/1/1)	9 (1/1/1)	11 (1/1/1)	11 (1/1/1)	13 (1/2/1)
−(An)	8 (1/1/0)	8 (1/1/0)	10 (1/1/1)	10 (1/1/1)	10 (1/1/1)	12 (1/1/1)	12 (1/1/1)	14 (1/2/1)
(d_{16},An) or (d_{16},PC)	9 (1/2/0)	9 (1/2/0)	11 (1/2/1)	11 (1/2/1)	11 (1/2/1)	13 (1/2/1)	13 (1/2/1)	15 (1/3/1)
(xxx).W	8 (1/2/0)	8 (1/2/0)	10 (1/2/1)	10 (1/2/1)	10 (1/2/1)	12 (1/2/1)	12 (1/2/1)	14 (1/3/1)
(xxx).L	10 (1/2/0)	10 (1/2/0)	12 (1/2/1)	12 (1/2/1)	12 (1/2/1)	14 (1/2/1)	14 (1/2/1)	16 (1/3/1)
(d_8,An,Xn) or d_8,PC,Xn)	11 (1/2/0)	11 (1/2/0)	13 (1/2/1)	13 (1/2/1)	13 (1/2/1)	15 (1/2/1)	15 (1/2/1)	17 (1/3/1)
(d_{16},An,Xn) or (d_{16},PC,Xn)	12 (1/2/0)	12 (1/2/0)	14 (1/2/1)	14 (1/2/1)	14 (1/2/1)	16 (1/2/1)	16 (1/2/1)	18 (1/3/1)
(B)	12 (1/2/0)	12 (1/2/0)	14 (1/2/1)	14 (1/2/1)	14 (1/2/1)	16 (1/2/1)	16 (1/2/1)	18 (1/3/1)
(d_{16},B)	15 (1/2/0)	15 (1/2/0)	17 (1/2/1)	17 (1/2/1)	17 (1/3/1)	19 (1/2/1)	19 (1/2/1)	21 (1/3/1)
(d_{32},B)	19 (1/3/0)	19 (1/3/0)	21 (1/3/1)	21 (1/3/1)	21 (1/3/1)	23 (1/3/1)	23 (1/3/1)	25 (1/4/1)
([B],I)	16 (2/2/0)	16 (2/2/0)	18 (2/2/1)	18 (2/2/1)	18 (2/2/1)	20 (2/2/1)	20 (2/2/1)	22 (2/3/1)
([B],I,d_{16})	19 (2/2/0)	19 (2/2/0)	21 (2/2/1)	21 (2/2/1)	21 (2/2/1)	23 (2/2/1)	23 (2/2/1)	25 (2/3/1)
([B],I,D_{32})	20 (2/3/0)	20 (2/3/0)	22 (2/3/1)	22 (2/3/1)	22 (2/3/1)	24 (2/3/1)	24 (2/3/1)	26 (2/4/1)
([d_{16},B],I)	19 (2/2/0)	19 (2/2/0)	21 (2/2/1)	21 (2/2/1)	21 (2/2/1)	23 (2/2/1)	23 (2/2/1)	25 (2/3/1)
([d_{16},B],I,d_{16})	22 (2/3/0)	22 (2/3/0)	24 (2/3/1)	24 (2/3/1)	24 (2/3/1)	26 (2/3/1)	26 (2/3/1)	28 (2/4/1)
([d_{16},B],d_{32})	23 (2/3/0)	23 (2/3/0)	25 (2/3/1)	25 (2/3/1)	25 (2/3/1)	27 (2/3/1)	27 (2/3/1)	29 (2/4/1)
([d_{32},B],I)	23 (2/3/0)	23 (2/3/0)	25 (2/3/1)	25 (2/3/1)	25 (2/3/1)	27 (2/3/1)	27 (2/3/1)	29 (2/4/1)
([d_{32},B],I,d_{15})	25 (2/3/0)	25 (2/3/0)	27 (2/3/1)	27 (2/3/1)	27 (2/3/1)	29 (2/3/1)	29 (2/3/1)	31 (2/4/1)
([d_{32},B],I,d_{32})	27 (2/4/0)	27 (2/4/0)	29 (2/4/1)	29 (2/4/1)	29 (2/4/1)	31 (2/4/1)	31 (2/4/1)	33 (2/5/1)

WORST CASE (Continued)

Source Address Mode	Destination							
	(d$_{8}$,An,Xn)	(d$_{16}$,An,Xn)	(B)	(d$_{16}$,B)	(d$_{32}$,B)	([B],I)	([B],I,d$_{16}$)	([B],I,D$_{32}$)
Rn	9 (0/1/1)	12 (0/2/1)	10 (0/1/1)	14 (0/2/1)	19 (0/2/1)	14 (1/1/1)	17 (1/2/1)	20 (1/2/1)
#<data>.B,W	9 (0/1/1)	12 (0/2/1)	10 (0/1/1)	14 (0/2/1)	19 (0/2/1)	14 (1/1/1)	17 (1/2/1)	20 (1/2/1)
#<data>.L	11 (0/1/1)	14 (0/2/1)	12 (0/1/1)	16 (0/2/1)	21 (0/2/1)	16 (1/1/1)	19 (1/2/1)	22 (1/2/1)
(An)	11 (1/1/1)	14 (1/2/1)	12 (1/1/1)	16 (1/2/1)	21 (1/2/1)	12 (2/1/1)	19 (2/2/1)	22 (2/2/1)
(An)+	11 (1/1/1)	14 (1/2/1)	12 (1/1/1)	16 (1/2/1)	21 (1/2/1)	12 (2/1/1)	19 (2/2/1)	22 (2/2/1)
−(An)	12 (1/1/1)	15 (1/2/1)	13 (1/1/1)	17 (1/2/1)	22 (1/2/1)	13 (2/1/1)	20 (2/2/1)	23 (2/2/1)
(d$_{16}$,An) or (d$_{16}$,PC)	13 (1/2/1)	16 (1/3/1)	14 (1/2/1)	18 (1/3/1)	23 (1/3/1)	14 (2/2/1)	21 (2/3/1)	24 (2/3/1)
(xxx).W	12 (1/2/1)	15 (1/3/1)	13 (1/2/1)	17 (1/3/1)	22 (1/3/1)	13 (2/2/1)	20 (2/3/1)	23 (2/3/1)
(xxx).L	14 (1/2/1)	17 (1/3/1)	15 (1/2/1)	19 (1/3/1)	24 (1/3/1)	15 (2/2/1)	22 (2/3/1)	25 (2/3/1)
(d$_{8}$,An,Xn) or (d$_{8}$,PC,Xn)	15 (1/2/1)	18 (1/3/1)	16 (1/2/1)	20 (1/3/1)	25 (1/3/1)	16 (2/2/1)	23 (2/3/1)	26 (2/3/1)
(d$_{16}$,An,Xn) or (d$_{16}$,PC,Xn)	16 (1/2/1)	19 (1/3/1)	17 (1/2/1)	21 (1/3/1)	26 (1/3/1)	17 (2/2/1)	24 (2/3/1)	27 (2/3/1)
(B)	16 (1/2/1)	19 (1/3/1)	17 (1/2/1)	21 (1/3/1)	26 (1/3/1)	17 (2/2/1)	24 (2/3/1)	27 (2/3/1)
(d$_{16}$,B)	19 (1/2/1)	22 (1/3/1)	20 (1/2/1)	24 (1/3/1)	29 (1/3/1)	20 (2/2/1)	27 (2/3/1)	30 (2/3/1)
(d$_{32}$,B)	23 (1/3/1)	26 (1/4/1)	24 (1/3/1)	28 (1/4/1)	33 (1/4/1)	24 (2/3/1)	31 (2/4/1)	34 (2/4/1)
([B],I)	20 (2/2/1)	23 (2/3/1)	21 (2/2/1)	25 (2/3/1)	30 (2/3/1)	21 (3/2/1)	28 (3/3/1)	31 (3/3/1)
([B],I,D$_{16}$)	23 (2/2/1)	26 (2/3/1)	24 (2/2/1)	28 (2/3/1)	33 (2/3/1)	24 (3/2/1)	31 (3/3/1)	34 (3/3/1)
([B],I,d$_{32}$)	24 (2/3/1)	27 (2/4/1)	25 (2/3/1)	29 (2/4/1)	34 (2/4/1)	25 (3/3/1)	32 (3/4/1)	35 (3/4/1)
([d$_{16}$,B],I)	23 (2/2/1)	26 (2/3/1)	24 (2/2/1)	28 (2/3/1)	33 (2/3/1)	24 (3/2/1)	31 (3/3/1)	34 (3/3/1)
([d$_{16}$,B],I,d$_{16}$)	26 (2/3/1)	29 (2/4/1)	27 (2/3/1)	31 (2/4/1)	36 (2/4/1)	27 (3/3/1)	34 (3/4/1)	37 (3/4/1)
([d$_{16}$,B],I,d$_{32}$)	27 (2/3/1)	30 (2/4/1)	28 (2/3/1)	32 (2/4/1)	37 (2/4/1)	28 (3/3/1)	35 (3/4/1)	38 (3/4/1)
([d$_{32}$,B],I)	27 (2/3/1)	30 (2/4/1)	28 (2/3/1)	32 (2/4/1)	37 (2/4/1)	28 (3/3/1)	35 (3/4/1)	38 (3/4/1)
([d$_{32}$,B],I,d$_{16}$)	29 (2/3/1)	32 (2/4/1)	30 (2/3/1)	34 (2/4/1)	39 (2/4/1)	30 (3/3/1)	37 (3/4/1)	40 (3/4/1)
([d$_{32}$,B],I,d$_{32}$)	31 (2/4/1)	34 (2/5/1)	32 (2/4/1)	36 (2/5/1)	41 (2/5/1)	32 (3/4/1)	39 (3/5/1)	42 (3/5/1)

WORST CASE (Concluded)

Source Address Mode	Destination					
	([d_{16},B],I)	([d_{16},B],I,d_{16})	([d_{16},B],I,d_{32})	([d_{32},B],I)	([d_{32},B],I,d_{16})	([d_{32},B],I,d_{32})
Rn	17 (1/2/1)	20 (1/2/1)	23 (1/3/1)	22 (1/2/1)	25 (1/3/1)	27 (1/3/1)
#<data>.B,W	17 (1/2/1)	20 (1/2/1)	23 (1/3/1)	22 (1/2/1)	25 (1/3/1)	27 (1/3/1)
#<data>.L	19 (1/2/1)	22 (1/2/1)	25 (1/3/1)	24 (1/2/1)	27 (1/3/1)	29 (1/3/1)
(An)	19 (2/2/1)	22 (2/2/1)	25 (2/3/1)	24 (2/2/1)	27 (2/3/1)	29 (2/3/1)
(An)+	19 (2/2/1)	22 (2/2/1)	25 (2/3/1)	24 (2/2/1)	27 (2/3/1)	29 (2/3/1)
−(An)	20 (2/2/1)	23 (2/2/1)	26 (2/3/1)	25 (2/2/1)	28 (2/3/1)	30 (2/3/1)
(d_{16},An) or (d_{16},PC)	21 (2/3/1)	24 (2/3/1)	27 (2/4/1)	26 (2/3/1)	29 (2/4/1)	31 (2/4/1)
(xxx).W	20 (2/3/1)	23 (2/3/1)	26 (2/4/1)	27 (2/3/1)	28 (2/4/1)	30 (2/4/1)
(xxx).L	22 (2/3/1)	25 (2/3/1)	28 (2/4/1)	29 (2/3/1)	30 (2/4/1)	32 (2/4/1)
(d_8,An,Xn) or (d_8,PC,Xn)	23 (2/3/1)	26 (2/3/1)	29 (2/4/1)	30 (2/3/1)	31 (2/4/1)	33 (2/4/1)
(d_{16},An,Xn) or (d_{16},PC,Xn)	24 (2/3/1)	27 (2/3/1)	30 (2/4/1)	31 (2/3/1)	32 (2/4/1)	34 (2/4/1)
(B)	24 (2/3/1)	27 (2/3/1)	30 (2/4/1)	31 (2/3/1)	32 (2/4/1)	34 (2/4/1)
(d_{16},B)	27 (2/3/1)	30 (2/3/1)	33 (2/4/1)	34 (2/3/1)	35 (2/4/1)	37 (2/4/1)
(d_{32},B)	31 (2/4/1)	34 (2/4/1)	37 (2/5/1)	38 (2/4/1)	39 (2/5/1)	41 (2/5/1)
([B],I)	28 (3/3/1)	31 (3/3/1)	34 (3/4/1)	35 (3/3/1)	36 (3/4/1)	38 (3/4/1)
([B],I,d_{16})	31 (3/3/1)	34 (3/3/1)	37 (3/4/1)	38 (3/3/1)	39 (3/4/1)	41 (3/4/1)
([B],I,d_{32})	32 (3/4/1)	35 (3/4/1)	38 (3/5/1)	39 (3/4/1)	40 (3/5/1)	42 (3/5/1)
([d_{16},B],I)	31 (3/3/1)	34 (3/3/1)	37 (3/4/1)	38 (3/3/1)	39 (3/4/1)	41 (3/4/1)
([d_{16},B],I,d_{16})	34 (3/4/1)	37 (3/4/1)	40 (3/5/1)	41 (3/4/1)	42 (3/5/1)	44 (3/5/1)
([d_{16},B],I,d_{32})	35 (3/4/1)	38 (3/4/1)	41 (3/5/1)	42 (3/4/1)	43 (3/5/1)	45 (3/5/1)
([d_{32},B],I)	35 (3/4/1)	38 (3/4/1)	41 (3/5/1)	42 (3/4/1)	43 (3/5/1)	45 (3/5/1)
([d_{32},B],I,d_{16})	37 (3/4/1)	40 (3/4/1)	43 (3/5/1)	44 (3/4/1)	45 (3/5/1)	47 (3/5/1)
([d_{32},B],I,d_{32})	39 (3/5/1)	42 (3/5/1)	45 (3/6/1)	46 (3/5/1)	47 (3/6/1)	49 (3/6/1)

10.2.7 Special Purpose MOVE Instruction

The special purpose MOVE timing table indicates the number of clock periods needed for the processor to fetch, calculate, and perform the special purpose MOVE operation on the control registers or specified effective address. The total number of clock cycles is outside the parentheses, the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Instruction			Best Case	Cache Case	Worst Case
	EXG	Ry,Rx	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
	MOVEC	Cr,Rn	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
	MOVEC	Rn,Cr	9 (0/0/0)	12 (0/0/0)	13 (0/1/0)
	MOVE	PSW,Rn	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
#	MOVE	PSW,Mem	5 (0/0/1)	5 (0/0/1)	7 (0/1/1)
*	MOVE	EA,CCR	4 (0/0/0)	4 (0/0/0)	5 (0/1/0)
*	MOVE	EA,SR	8 (0/0/0)	8 (0/0/0)	11 (0/2/0)
‡	MOVEM	EA,RL	8+4n (n/0/0)	8+4n (n/0/0)	9+4n (n/1/0)
‡	MOVEM	RL,EA	4+3n (0/0/n)	4+3n (0/0/n)	5+3n (0/1/n)
	MOVEP.W	Dn,(d_{16},An)	8 (0/0/2)	11 (0/0/2)	11 (0/1/2)
	MOVEP.L	Dn,(d_{16},An)	14 (0/0/4)	17 (0/0/4)	17 (0/1/4)
	MOVEP.W	(d_{16},An),Dn	10 (2/0/0)	12 (2/0/0)	12 (2/1/0)
	MOVEP.L	(d_{16},An),Dn	16 (4/0/0)	18 (4/0/0)	18 (4/1/0)
‡	MOVES	EA,Rn	7 (1/0/0)	7 (1/0/0)	8 (1/1/0)
‡	MOVES	Rn,EA	5 (0/0/1)	5 (0/0/1)	7 (0/1/1)
	MOVE	USP	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
	SWAP	Rx,Ry	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)

n = Number of Registers to Transfer
RL = Register List
* Add Fetch Effective Address Time
# Add Calculate Effective Address Time
‡ Add Calculate Immediate Address Time

10.2.8 Arithmetic/Logical Instructions

The arithmetic/logical operations timing table indicates the number of clock periods needed for the processor to perform the specified arithmetic/logical operation using the specified addressing mode. It also includes, in worst case, the amount of time needed to prefetch the next instruction. Footnotes specify when to add either fetch address or fetch immediate effective address time. This sum gives the total effective execution time for the operation using the specified addressing mode. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Instruction			Best Case	Cache Case	Worst Case
*	ADD	EA,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	ADDA	EA,An	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	ADD	Dn,EA	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
*	AND	EA,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	AND	Dn,EA	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
*	EOR	Dn,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	EOR	Dn,Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
*	OR	EA,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	OR	Dn,EA	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
*	SUB	EA,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	SUBA	EA,An	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	SUB	Dn,EA	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
*	CMP	EA,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	CMPA	EA,An	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
**	CMP2	EA,Rn	16 (1/0/0)	18 (1/0/0)	18 (1/1/0)
*	MUL.W	EA,Dn	25 (0/0/0)	27 (0/0/0)	28 (0/1/0)
**	MUL.L	EA,Dn	41 (0/0/0)	43 (0/0/0)	44 (0/1/0)
*	DIVU.W	EA,Dn	42 (0/0/0)	44 (0/0/0)	44 (0/1/0)
**	DIVU.L	EA,Dn	76 (0/0/0)	78 (0/0/0)	79 (0/1/0)
*	DIVS.W	EA,Dn	54 (0/0/0)	56 (0/0/0)	57 (0/1/0)
**	DIVS.L	EA,Dn	88 (0/0/0)	90 (0/0/0)	91 (0/1/0)

*Add Fetch Effective Address Time
**Add Fetch Immediate Address Time

10.2.9 Immediate Arithmetic/Logical Instructions

The immediate arithmetic/logical operations timing table indicates the number of clock periods needed for the processor to fetch the source immediate data value, and perform the specified arithmetic/logical operation using the specified destination addressing mode. Footnotes indicate when to add appropriate fetch effective or fetch immediate effective address time. This computation will give the total execution time needed to perform the appropriate immediate arithmetic/logical operation. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

	Instruction		Best Case	Cache Case	Worst Case
	MOVEQ	#<data>,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
	ADDQ	#<data>,Rn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	ADDQ	#<data>,Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
	SUBQ	#<data>,Rn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	SUBQ	#<data>,Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
**	ADDI	#<data>,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
**	ADDI	#<data>,Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
**	ANDI	#<data>,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
**	ANDI	#<data>,Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
**	EORI	#<data>,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
**	EORI	#<data>,Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
**	ORI	#<data>,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
**	ORI	#<data>,Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
**	SUBI	#<data>,Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
**	SUBI	#<data>,Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
**	CMPI	#<data>,EA	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)

*Add Fetch Effective Address Time
**Add Fetch Immediate Address Time

10.2.10 Binary Coded Decimal Operations

The binary coded decimal operations table indicates the number of clock periods needed for the processor to perform the specified operation using the given addressing modes, with complete execution times given. No additional tables are needed to calculate total effective execution time for these instructions. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Instruction		Best Case	Cache Case	Worst Case
ABCD	Dn,Dn	4 (0/0/0)	4 (0/0/0)	5 (0/1/0)
ABCD	−(An),−(An)	14 (2/0/1)	16 (2/0/1)	17 (2/1/1)
SBCD	Dn,Dn	4 (0/0/0)	4 (0/0/0)	5 (0/1/0)
SBCD	−(An),−(An)	14 (2/0/1)	16 (2/0/1)	17 (2/1/1)
ADDX	Dn,Dn	2 (0/0/0)	2 (0/0/0)	3 (0/1/0)
ADDX	−(An),−(An)	10 (2/0/1)	12 (2/0/1)	13 (2/1/1)
SUBX	Dn,Dn	2 (0/0/0)	2 (0/0/0)	3 (0/1/0)
SUBX	−(An),−(An)	10 (2/0/1)	12 (2/0/1)	13 (2/1/1)
CMPM	(An)+,(An)+	8 (2/0/0)	9 (2/0/0)	10 (2/1/0)
PACK	Dn,Dn,#<data>	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
PACK	−(An),−(An),#<data>	11 (1/0/1)	13 (1/0/1)	13 (1/1/1)
UNPK	Dn,Dn,#<data>	5 (0/0/0)	8 (0/0/0)	9 (0/1/0)
UNPK	−(An),−(An),#<data>	11 (1/0/1)	13 (1/0/1)	13 (1/1/1)

10.2.11 Single Operand Instructions

The single operand instructions table indicates the number of clock periods needed for the processor to perform the specified operation on the given addressing mode. Footnotes indicate when it is necessary to add another table entry to calculate the total effective execution time for the instruction. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

	Instruction		Best Case	Cache Case	Worst Case
	CLR	Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
#	CLR	Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
	NEG	Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	NEG	Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
	NEGX	Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	NEGX	Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
	NOT	Dn	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
*	NOT	Mem	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
	EXT	Dn	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
	NBCD	Dn	6 (0/0/0)	6 (0/0/0)	6 (0/1/0)
	Scc	Dn	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
#	Scc	Mem	6 (0/0/1)	6 (0/0/1)	6 (0/1/1)
	TAS	Dn	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
#	TAS	Mem	12 (1/0/1)	12 (1/0/1)	13 (1/1/1)
*	TST	EA	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)

*Add Fetch Effective Address Time
#Add Calculate Effective Address Time

10.2.12 Shift/Rotate Instructions

The shift/rotate instructions table indicates the number of clock periods needed for the processor to perform the specified operation on the given addressing mode. Footnotes indicate when it is necessary to add another table entry to calculate the total effective execution time for the instruction. The number of bits shifted does not affect execution time. The total number of clock cycles is outside the parentheses, the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Instruction			Best Case	Cache Case	Worst Case
	LSL	Dn (Static)	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
	LSR	Dn (Static)	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
	LSL	Dn (Dynamic)	3 (0/0/0)	6 (0/0/0)	6 (0/1/0)
	LSR	Dn (Dynamic)	3 (0/0/0)	6 (0/0/0)	6 (0/1/0)
*	LSL	Mem by 1	5 (0/0/1)	5 (0/0/1)	6 (0/1/1)
*	LSR	Mem by 1	5 (0/0/1)	5 (0/0/1)	6 (0/1/1)
	ASL	Dn	5 (0/0/0)	8 (0/0/0)	8 (0/1/0)
	ASR	Dn	3 (0/0/0)	6 (0/0/0)	6 (0/1/0)
*	ASL	Mem by 1	6 (0/0/1)	6 (0/0/1)	7 (0/1/1)
*	ASR	Mem by 1	5 (0/0/1)	5 (0/0/1)	6 (0/1/1)
	ROL	Dn	5 (0/0/0)	8 (0/0/0)	8 (0/1/0)
	ROR	Dn	5 (0/0/0)	8 (0/0/0)	8 (0/1/0)
*	ROL	Mem by 1	7 (0/0/1)	7 (0/0/1)	7 (0/1/1)
*	ROR	Mem by 1	7 (0/0/1)	7 (0/0/1)	7 (0/1/1)
	ROXL	Dn	9 (0/0/0)	12 (0/0/0)	12 (0/1/0)
	ROXR	Dn	9 (0/0/0)	12 (0/0/0)	12 (0/1/0)
*	ROXd	Mem by 1	5 (0/0/1)	5 (0/0/1)	6 (0/1/1)

*Add Fetch Effective Address Time
d Direction of Shift/Rotate; L or R

10.2.13 Bit Manipulation Instructions

The bit manipulation instructions table indicates the number of clock periods needed for the processor to perform the specified bit operation on the given addressing mode. Footnotes indicate when it is necessary to add another table entry to calculate the total effective execution time for the instruction. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Instruction			Best Case	Cache Case	Worst Case
	BTST	#<data>,Dn	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
	BTST	Dn,Dn	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
**	BTST	#<data>,Mem	4 (0/0/0)	4 (0/0/0)	5 (0/1/0)
*	BTST	Dn,Mem	4 (0/0/0)	4 (0/0/0)	5 (0/1/0)
	BCHG	#<data>,Dn	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
	BCHG	Dn,Dn	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
**	BCHG	#<data>,Mem	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
*	BCHG	Dn,Mem	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
	BCLR	#<data>,Dn	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
	BCLR	Dn,Dn	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
**	BCLR	#<data>,Mem	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
*	BCLR	Dn,Mem	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
	BSET	#<data>,Dn	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
	BSET	Dn,Dn	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
**	BSET	#<data>,Mem	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
*	BSET	Dn,Mem	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)

*Add Fetch Effective Address Time
**Add Fetch Immediate Address Time

10.2.14 Bit Field Manipulation Instructions

The bit field manipulation instructions table indicates the number of clock periods needed for the processor to perform the specified bit field operation using the given addressing mode. Footnotes indicate when it is necessary to add another table entry to calculate the total effective execution time for the instruction. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

	Instruction		Best Case	Cache Case	Worst Case
	BFTST	Dn	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
‡	BFTST	Mem (<5 Bytes)	11 (1/0/0)	11 (1/0/0)	12 (1/1/0)
‡	BFTST	Mem (5 Bytes)	15 (2/0/0)	15 (2/0/0)	16 (2/1/0)
	BFCHG	Dn	9 (0/0/0)	12 (0/0/0)	12 (0/1/0)
‡	BFCHG	Mem (<5 Bytes)	16 (1/0/1)	16 (1/0/1)	16 (1/1/1)
‡	BFCHG	Mem (5 Bytes)	24 (2/0/2)	24 (2/0/2)	24 (2/1/2)
	BFCLR	Dn	9 (0/0/0)	12 (0/0/0)	12 (0/1/0)
‡	BFCLR	Mem (<5 Bytes)	16 (1/0/1)	16 (1/0/1)	16 (1/1/1)
‡	BFCLR	Mem (5 Bytes)	24 (2/0/2)	24 (2/0/2)	24 (2/1/2)
	BFSET	Dn	9 (0/0/0)	12 (0/0/0)	12 (0/1/0)
‡	BFSET	Mem (<5 Bytes)	16 (1/0/1)	16 (1/0/1)	16 (1/1/1)
‡	BFSET	Mem (5 Bytes)	24 (2/0/2)	24 (2/0/2)	24 (2/1/2)
	BFEXTS	Dn	5 (0/0/0)	8 (0/0/0)	8 (0/1/0)
‡	BFEXTS	Mem (<5 Bytes)	13 (1/0/0)	13 (1/0/0)	13 (1/1/0)
‡	BFEXTS	Mem (5 Bytes)	18 (2/0/0)	18 (2/0/0)	18 (2/1/0)
	BFEXTU	Dn	5 (0/0/0)	8 (0/0/0)	8 (0/1/0)
‡	BFEXTU	Mem (<5 Bytes)	13 (1/0/0)	13 (1/0/0)	13 (1/1/0)
‡	BFEXTU	Mem (5 Bytes)	18 (2/0/0)	18 (2/0/0)	18 (2/1/0)
	BFINS	Dn	7 (0/0/0)	10 (0/0/0)	10 (0/1/0)
‡	BFINS	Mem (<5 Bytes)	14 (1/0/1)	14 (1/0/1)	15 (1/1/1)
‡	BFINS	Mem (5 Bytes)	20 (2/0/2)	20 (2/0/2)	21 (2/1/2)
	BFFFO	Dn	15 (0/0/0)	18 (0/0/0)	18 (0/1/0)
‡	BFFFO	Mem (<5 Bytes)	24 (1/0/0)	24 (1/0/0)	24 (1/1/0)
‡	BFFFO	Mem (5 Bytes)	32 (2/0/0)	32 (2/0/0)	32 (2/1/0)

‡ Add Calculate Immediate Address Time

NOTE: A bit field of 32 bits may span 5 bytes that requires two operand cycles to access or may span 4 bytes that requires only one operand cycle to access.

10.2.15 Conditional Branch Instructions

The conditional branch instructions table indicates the number of clock periods needed for the processor to perform the specified branch on the given branch size, with complete execution times given. No additional tables are needed to calculate total effective execution time for these instructions. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Instruction	Best Case	Cache Case	Worst Case
Bcc (Taken)	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)
Bcc.B (Not Taken)	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
Bcc.W (Not Taken)	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
Bcc.L (Not Taken)	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)
DBcc (cc = False, Count Not Expired)	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)
DBcc (cc = False, Count Expired)	7 (0/0/0)	10 (0/0/0)	10 (0/3/0)
DBcc (cc = True)	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)

10.2.16 Control Instructions

The control instructions table indicates the number of clock periods needed for the processor to perform the specified operation. Footnotes specify when it is necessary to add an entry from another table to calculate the total effective execution time for the given instruction. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Instruction		Best Case	Cache Case	Worst Case
	ANDI to SR	9 (0/0/0)	12 (0/0/0)	15 (0/2/0)
	EORI to SR	9 (0/0/0)	12 (0/0/0)	15 (0/2/0)
	ORI to SR	9 (0/0/0)	12 (0/0/0)	15 (0/2/0)
	ANDI to CCR	9 (0/0/0)	12 (0/0/0)	15 (0/2/0)
	EORI to CCR	9 (0/0/0)	12 (0/0/0)	15 (0/2/0)
	ORI to CCR	9 (0/0/0)	12 (0/0/0)	15 (0/2/0)
	BSR	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
**	CALLM (Type 0)	28 (2/0/6)	30 (2/0/6)	36 (2/2/6)
**	CALLM (Type 1) — No Stack Copy —	48 (5/0/8)	50 (5/0/8)	56 (5/2/8)
**	CALLM (Type 1) — No Stack Copy —	55 (6/0/8)	57 (6/0/8)	64 (6/2/8)
**	CALLM (Type 1) — STACK COPY —	63 + 6n (7 + n/0/8 + n)	65 + 6n (7 + n/0/8 + n)	71 + 6n (7 + n/2/8 + n)
‡	CAS (Successful Compare)	15 (1/0/1)	15 (1/0/1)	16 (1/1/1)
‡	CAS (Unsuccessful Compare)	12 (1/0/0)	12 (1/0/0)	13 (1/1/0)
	CAS2 (Successful Compare)	23 (2/0/2)	25 (2/0/2)	28 (2/2/2)
	CAS2 (Unsuccessful Compare)	19 (2/0/0)	22 (2/0/0)	25 (2/2/0)
*	CHK	8 (0/0/0)	8 (0/0/0)	8 (0/1/0)
**	CHK2 EA,Rn	16 (2/0/0)	18 (2/0/0)	18 (2/1/0)
%	JMP	1 (0/0/0)	4 (0/0/0)	7 (0/2/0)
%	JSR	3 (0/0/1)	5 (0/0/1)	11 (0/2/1)
#	LEA	2 (0/0/0)	2 (0/0/0)	3 (0/1/0)
	LINK.W	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
	LINK.L	4 (0/0/1)	6 (0/0/1)	10 (0/2/1)
	NOP	2 (0/0/0)	2 (0/0/0)	3 (0/1/0)
#	PEA	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
	RTD	9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
	RTM (Type 0)	18 (4/0/0)	19 (4/0/0)	22 (4/2/0)
	RTM (Type 1)	31 (6/0/1)	32 (6/0/1)	35 (6/2/1)
	RTR	13 (2/0/0)	14 (2/0/0)	15 (2/2/0)
	RTS	9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
	UNLK	5 (1/0/0)	6 (1/0/0)	7 (1/1/0)

n Number of Operand Transfers Required
* Add Fetch Effective Address Time
# Add Calculate Effective Address Time
% Add Jump Effective Address Time
**Add Fetch Immediate Address Time
‡ Add Calculate Immediate Address Time

10.2.17 Exception-Related Instructions

The exception-related instructions table indicates the number of clock periods needed for the processor to perform the specified exception-related action. Footnotes specify when it is necessary to add the entry from another table to calculate the total effective execution time for the given instruction. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Instruction	Best Case	Cache Case	Worst Case
BKPT	9 (1/0/0)	10 (1/0/0)	10 (1/0/0)
Interrupt (I-Stack)	26 (2/0/4)	26 (2/0/4)	33 (2/2/4)
Interrupt (M-Stack)	41 (2/0/8)	41 (2/0/8)	48 (2/2/8)
RESET Instruction	518 (0/0/0)	518 (0/0/0)	519 (0/1/0)
STOP	8 (0/0/0)	8 (0/0/0)	8 (0/0/0)
Trace	25 (1/0/5)	25 (1/0/5)	32 (1/2/5)
TRAP #n	20 (1/0/4)	20 (1/0/4)	27 (1/2/4)
Illegal Instruction	20 (1/0/4)	20 (1/0/4)	27 (1/2/4)
A-Line Trap	20 (1/0/4)	20 (1/0/4)	27 (1/2/4)
F-Line Trap	20 (1/0/4)	20 (1/0/4)	27 (1/2/4)
Privilege Violation	20 (1/0/4)	20 (1/0/4)	27 (1/2/4)
TRAPcc (Trap)	23 (1/0/5)	25 (1/0/5)	32 (1/2/5)
TRAPcc (No Trap)	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
TRAPcc.W (Trap)	23 (1/0/5)	25 (1/0/5)	33 (1/3/5)
TRAPcc.W (No Trap)	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
TRAPcc.L (Trap)	23 (1/0/5)	25 (1/0/5)	33 (1/3/5)
TRAPcc.L (No Trap)	5 (0/0/0)	8 (0/0/0)	10 (0/2/0)
TRAPV (Trap)	23 (1/0/5)	25 (1/0/5)	32 (1/2/5)
TRAPV (No Trap)	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)

10.2.18 Save and Restore Operations

The save and restore operations table indicates the number of clock periods needed for the processor to perform the specified state save or return from exception, with complete execution times and stack length given. No additional tables are needed to calculate total effective execution time for these operations. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles are given inside the parentheses as (r/p/w). They are included in the total clock cycle number.

Operation	Best Case	Cache Case	Worst Case
Bus Cycle Fault (Short)	42 (1/0/10)	43 (1/0/10)	50 (1/2/10)
Bus Cycle Fault (Long)	79 (1/0/24)	79 (1/0/24)	86 (1/2/24)
RTE (Normal)	20 (4/0/0)	21 (4/0/0)	24 (4/2/0)
RTE (Six-Word)	20 (4/0/0)	21 (4/0/0)	24 (4/2/0)
RTE (Throwaway)*	15 (4/0/0)	16 (4/0/0)	39 (4/0/0)
RTE (Coprocessor)	31 (7/0/0)	32 (7/0/0)	33 (7/1/0)
RTE (Short Fault)	42 (10/0/0)	43 (10/0/0)	45 (10/2/0)
RTE (Long Fault)	91 (24/0/0)	92 (24/0/0)	94 (24/2/0)

*Add the time for RTE on second stack frame.

SECTION 11
APPLICATIONS INFORMATION

This section provides guidelines for using the MC68020. The section contains information on the following topics:

Module Support
Access Levels
Floating-Point Units

11.1 MODULE SUPPORT

The MC68020 includes support for modules with the call module (CALLM) and return from module (RTM) instructions. The CALLM instruction references a module descriptor. This descriptor contains control information for entry into the called module. The CALLM instruction creates a module stack frame and stores the current module state in that frame and loads a new module state from the referenced descriptor. The RTM instruction recovers the previous module state from the stack frame and returns to the calling module.

The module interface facilitates finer resolution of access control by external hardware. Although the MC68020 does not interpret the access control information, it does communicate with external hardware when the access control is to be changed and relies on the external hardware to verify that the changes are legal.

11.1.1 Module Descriptor

Figure 11-1 illustrates the format of the module descriptor. The first long word contains control information used during the execution of the CALLM instruction. The remaining

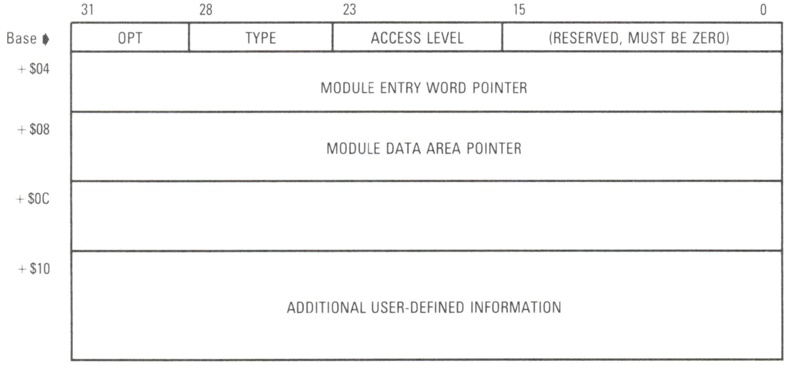

31
28
23
15
0
Base ➧
OPT
TYPE
ACCESS LEVEL
(RESERVED, MUST BE ZERO)
+$04
MODULE ENTRY WORD POINTER
+$08
MODULE DATA AREA POINTER
+$0C
+$10
ADDITIONAL USER-DEFINED INFORMATION

Figure 11-1. Module Descriptor Format

locations contain data which may be loaded into processor registers by the CALLM instruction.

The Opt field specifies how arguments are to be passed to the called module; the MC68020 recognizes only the options of 000 and 100; all others cause a format exception. The 000 option indicates that the called module expects to find arguments from the calling module on the stack just below the module stack frame. In cases where there is a change of stack pointer during the call, the MC68020 will copy the arguments from the old stack to the new stack. The 100 option indicates that the called module will access the arguments from the calling module through an indirect pointer in the stack of the calling module. Hence, the arguments are not copied, but the MC68020 puts the value of the stack pointer from the calling module in the module stack frame.

The Type field specifies the type of the descriptor; the MC68020 only recognizes descriptors of type $00 and $01, all others cause a format exception. The $00 type descriptor defines a module for which there is no change in access rights, and the called module builds its stack frame on top of the stack used by the calling module. The $01 type descriptor defines a module for which there may be a change in access rights; such a called module may have a separate stack area from that of the calling module.

The access level field is used only with the type $01 descriptor and is passed to external hardware to change the access control.

The module entry word pointer specifies the entry address of the called module. The first word at the entry address (see Figure 11-2) specifies the register to be saved in the module stack frame and then loaded with the module descriptor data area pointer; the first instruction of the module starts with the next word. The module descriptor data area pointer field contains the address of the called module data area.

15	14 13 12	11	10	9	8	7	6	5	4	3	3	1	0
D/A	REGISTER	0	0	0	0	0	0	0	0	0	0	0	0
OPERATION WORD OF FIRST INSTRUCTION													

Figure 11-2. Module Entry Word

If the access change requires a change of stack pointer, the old value is saved in the module stack frame, and the new value is taken from the module descriptor stack pointer field. Any further information in the module descriptor is user defined.

All module descriptor types $10–$1F are reserved for user definition and cause a format error exception. This provides the user with a means of disabling any module by setting a single bit in its descriptor without loss of any descriptor information.

If the called module does not wish the module data area pointer to be loaded into a register, the module entry word can select register A7, and the loaded value will be overwritten with the correction stack pointer value after the module stack frame is created and filled.

11.1.2 Module Stack Frame

Figure 11-3 illustrates the format of the module stack frame. This frame is constructed by the CALLM instruction and is removed by the RTM instruction. The first and second long words contain control information passed by the CALLM instruction to the RTM instruction. The module descriptor pointer contains the address of the descriptor used during the module call. All other locations contain information to be restored on return to the calling module.

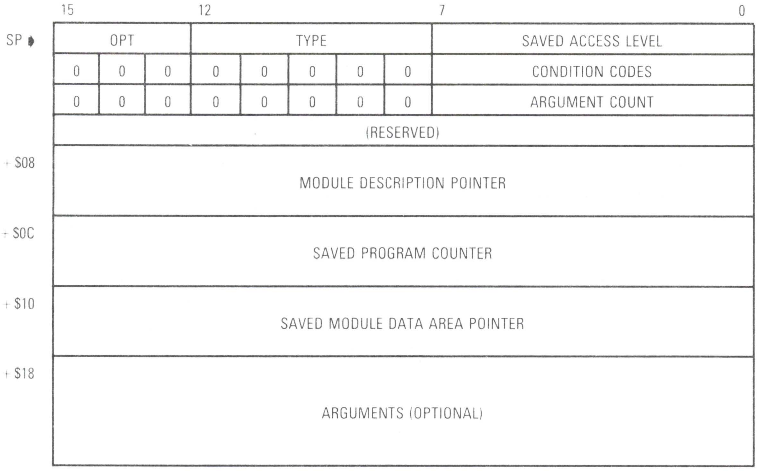

15
12
7
0
SP
OPT
TYPE
SAVED ACCESS LEVEL
0 0 0 0 0 0 0 0
CONDITION CODES
0 0 0 0 0 0 0 0
ARGUMENT COUNT
(RESERVED)
+ $08
MODULE DESCRIPTION POINTER
+ $0C
SAVED PROGRAM COUNTER
+ $10
SAVED MODULE DATA AREA POINTER
+ $18
ARGUMENTS (OPTIONAL)

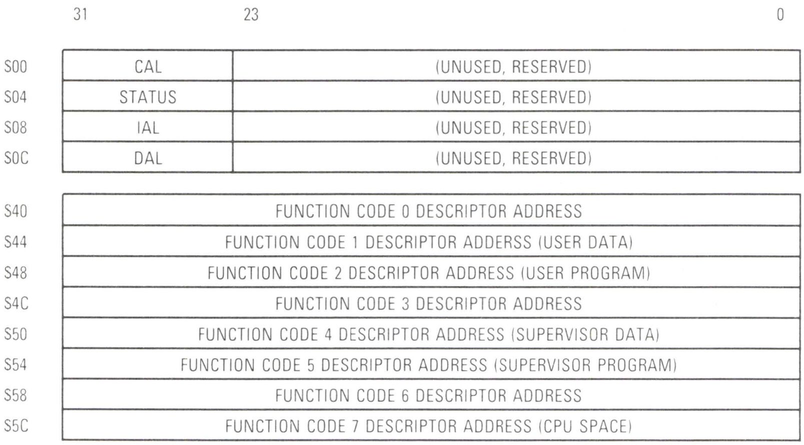
Figure 11-3. Module Call Stack Frame

The program counter is the saved address of the instruction following the CALLM instruction. The Opt and Type fields which specify the argument options and type of module stack frame, are copied to the frame from the module descriptor by the CALLM instruction; the RTM instruction will cause a format error if the Opt and Type fields do not have recognizable values. The access level is the saved access control information, which is saved from external hardware by the CALLM instruction and restored by the RTM instruction. The argument count field is set by the CALLM instruction and is used by the RTM instruction to remove arguments from the stack of the calling module. The contents of the CCR are saved by the CALLM instruction and restored by the RTM instruction. The saved stack pointer field contains the value of the stack pointer when the CALLM instruction started execution, and that value is restored by RTM. The saved module data pointer field contains the saved value of the module data area pointer register from the calling module.

11.2 ACCESS LEVELS

The MC68020 module mechanism supports a finer level of access control beyond the distinction between user and supervisor modes. The module mechanism allows a module with limited access rights to call a module with greater access rights. With the help of external hardware, the processor can verify that an increase in access rights is allowable or can detect attempts by a module to gain access rights to which it is not entitled.

Type $01 module descriptors and module stack frames indicate a request to change access levels. While processing a type $01 descriptor or frame, the CALLM and RTM instructions communicate with external access contol hardware via accesses in the CPU space. For these accesses, address bit [19–16] equal 0001. Figure 11-4 shows the address map for these CPU space accesses. If the processor receives a bus error on any of these CPU space accesses during the execution of a CALLM or RTM instruction, the processor will take a format error exception.



	31	23	0
$00	CAL	(UNUSED, RESERVED)	
$04	STATUS	(UNUSED, RESERVED)	
$08	IAL	(UNUSED, RESERVED)	
$0C	DAL	(UNUSED, RESERVED)	
$40	FUNCTION CODE 0 DESCRIPTOR ADDRESS		
$44	FUNCTION CODE 1 DESCRIPTOR ADDERSS (USER DATA)		
$48	FUNCTION CODE 2 DESCRIPTOR ADDRESS (USER PROGRAM)		
$4C	FUNCTION CODE 3 DESCRIPTOR ADDRESS		
$50	FUNCTION CODE 4 DESCRIPTOR ADDRESS (SUPERVISOR DATA)		
$54	FUNCTION CODE 5 DESCRIPTOR ADDRESS (SUPERVISOR PROGRAM)		
$58	FUNCTION CODE 6 DESCRIPTOR ADDRESS		
$5C	FUNCTION CODE 7 DESCRIPTOR ADDRESS (CPU SPACE)		

Figure 11-4. Access Level Control Bus Registers

The current access level register (CAL) contains the access level rights of the currently executing module. The increase access level register (IAL) is the register through which the processor requests increased access rights. The decrease access level register (DAL) is the register through which the processor requests decreased access rights. The formats of these three registers are undefined to the main processor, but the main processor assumes that information read from the module descriptor stack frame or the current access level register can be meaningfully written to the increase access level register or the decrease access level register. The access status register allows the processor to query the external hardware as to the legality of intended access level transitions. Table 11-1 lists the valid values of the access status register.

Table 11-1. Access Status Register Codes

Value	Validity	Processor Action
00	Invalid	Format Error
01	Valid	No Change in Access Rights
02–03	Valid	Change Access Rights with No Change of Stack Pointer
04–07	Valid	Change Access Rights and Change Stack Pointer
Other	Undefined	Undefined (Take Format Error Exception)

The processor uses the descriptor address registers during the CALLM instruction to communicate the address of the type $01 descriptor. This allows external hardware to verify that the address is a valid address for a type $01 descriptor. This prevents a module from creating a type $01 descriptor to surreptitiously increase its access rights.

11.2.1 Module Call

The CALLM instuction is used to make the module call. For the type $00 module descriptor, the processor simply creates and fills the module stack frame at the top of the active system stack. The condition codes of the calling module are saved in the CCR field of the frame. If Opt is equal to 000 (arguments passed on the stack) in the module descriptor, the MC68020 does not save the stack pointer or load a new stack pointer value. The processor uses the module entry word to save and load the module data area pointer register and then begins execution of the called module.

For the type $01 module descriptor, the processor must first obtain the current access level from extenal hardware. It also verifies that the calling module has the right to read from the area pointed to by the current value of the stack pointer by reading from that address. It passes the descriptor address and increase access level to external hardware for validation and then reads the access status. If external hardware determines that the change in access rights should not be granted, the access status is zero, and the processor takes a format error exception. No visible processor registers are changed, nor should the current access level enforced by external hardware be changed. If external hardware determines that a change should be granted, the external hardware changes its access level, and the processor proceeds. If the access status register indicates that a change in the stack pointer is required, the stack pointer is saved internally, a new value is loaded from the module descriptor, and arguments are copied from the calling stack to the new stack. Finally, the module stack frame is created and filled on the top of the current stack. The condition codes of the calling module are saved in the CCR field of the frame. Execution of the called module then begins as with a type $00 descriptor.

11.2.2 Module Return

The RTM instruction is used to return from a module. For the type $00 module stack frame, the processor reloads the condition codes, the program counter, and the module data area pointer register from the frame. The frame is removed from the top of the stack, the argument count is added to the stack pointer, and execution returns to the calling module.

For the type $01 module stack frame, the processor reads the access level, condition codes, program counter, saved module data area pointer, and saved stack pointer from the module stack frame. The access level is written to the decrease access level register for validation by external hardware; the processor then reads the access status to check the validation. If the external hardware determines that the change in access right should not be granted, the access status is zero, and the processor takes a format error exception. No visible processor registers are changed, nor should the current access level which is enforced by external hardware be changed. If the external hardware determines that the change in access rights should be granted, the external hardware changes its access level, the values read from the module stack frame are loaded into the corresponding processor registers, the argument count is added to the new stack pointer value, and execution returns to the calling module.

If the called module does not wish the saved module data pointer to be loaded into a register, the RTM instruction word can select register A7, and the loaded value will be overwritten with the correct stack pointer value after the module stack frame is deallocated.

11.3 FLOATING-POINT UNITS

Floating-point support for the MC68020 is provided by the MC68881 floating-point coprocessor and the MC68882 enhanced floating-point coprocessor. Both devices offer a full implementation of the *IEEE Standard for Binary Floating-Point Arithmetic* (754). The MC68882 is a pin and software-compatible upgrade of the MC68881, with an optimized MPU interface that provides over 1.5 times the performance of the MC68881 at the same clock frequency.

Both coprocessors provide a logical extension to the integer data processing capabilities of the main processor. They contain a very high-performance floating-point arithmetic unit and a set of floating-point data registers that are utilized in a manner that is analagous to the use of the integer data registers of the processor. The MC68881/MC68882 instruction set, which is a natural extension of all earlier members of the M68000 Family, and supports all of the addressing modes and data types of the host MC68020. The programmer perceives the MC68020/coprocessor execution model as if both devices are implemented on one chip. In addition to supporting the full IEEE standard, the MC68881 and MC68882 provide a full set of trigonometric and transcendental functions, on-chip constants, and a full 80-bit extended-precision real-data format.

The interface of the MC68020 to the MC68881 or the MC68882 is easily tailored to system cost/performance needs. The MC68020 and the MC68881/MC68882 communicate via standard asynchronous M68000 bus cycles. All data transfers are performed by the main processor at the request of the MC68881/MC68882; thus, memory management, bus errors, address errors, and bus arbitration function as if the MC68881/MC68882 instructions are executed by the main processor. The floating-point unit and the processor may operate at different clock speeds, and up to seven floating-point coprocessors may reside in an MC68020 system simultaneously.

Figure 11-5 illustrates the coprocessor interface connection of an MC68881/MC68882 to an MC68020 (uses entire 32-bit data bus). The MC68881/MC68882 is configured to operate with a 32-bit data bus when both the A0 and $\overline{\text{SIZE}}$ pins are connected to V_{CC}. Refer to the *MC68881/MC68882 User's Manual* for configuring the MC68881/MC68882 for smaller data bus widths.

The chip select ($\overline{\text{CS}}$) decode circuitry is asynchronous logic that detects when a particular floating-point coprocessor is addressed. The MC68020 signals used by the logic include the function code signals (FC2–FC0), and the address lines (A19–A13). Refer to **SECTION 9 COPROCESSOR INTERFACE DESCRIPTION** for more information concerning the encoding of these signals. All or just a subset of these lines may be decoded, depending on the number of coprocessors in the system and the degree of redundant mapping allowed in the system.

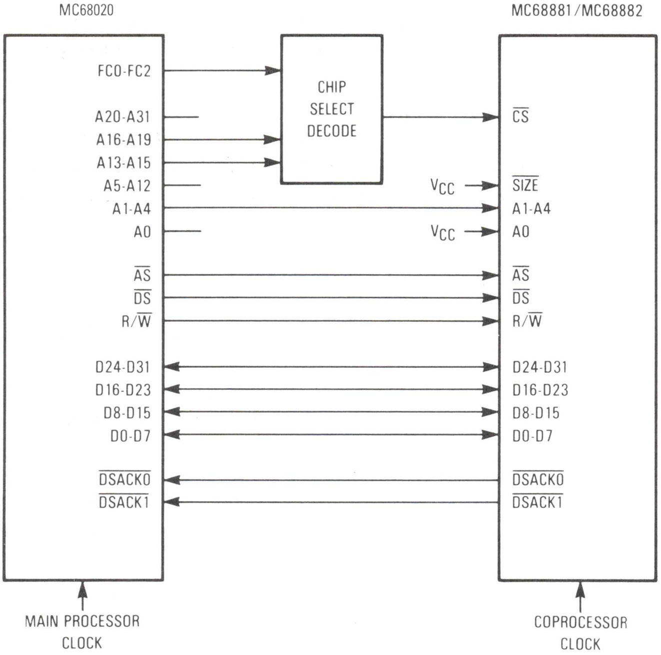

MC68020
MC68881/MC68882
FC0-FC2
A20-A31
A16-A19
A13-A15
A5-A12
A1-A4
A0
AS
DS
R/W
D24-D31
D16-D23
D8-D15
D0-D7
DSACK0
DSACK1
CHIP SELECT DECODE
VCC
CS
SIZE
A1-A4
A0
AS
DS
R/W
D24-D31
D16-D23
D8-D15
D0-D7
DSACK0
DSACK1
MAIN PROCESSOR CLOCK
COPROCESSOR CLOCK

Figure 11-5. 32-Bit Data Bus Coprocessor Connection

The major concern of a system designer is to design a $\overline{CS}$ interface that meets the ac electrical specifications for both the MC68020 (MPU) and the MC68881/MC68882 (FPCP) without adding unnecessary wait states to FPCP accesses. The following maximum specifications (relative to CLK low) meet these objectives:

$$t_{CLK} \text{ low to } \overline{AS} \text{ low} \leq (\text{MPU Spec 1} - \text{MPU Spec 47A} - \text{FPCP Spec 19}) \quad (1)$$

$$t_{CLK} \text{ low to } \overline{CS} \text{ low} \leq (\text{MPU Spec 1} - \text{MPU Spec 47A} - \text{FPCP Spec 19}) \quad (2)$$

Even though requirement (1) is not met under worst case conditions, if the MPU $\overline{AS}$ is loaded within specifications and the $\overline{AS}$ input to the FPCP is unbuffered, the requirement is met under typical conditions. Designing the $\overline{CS}$ generation circuit to meet requirement (2) provides the highest probability that accesses to the FPCP occur without unnecessary wait states. A PAL 16L8 (Figure 11-6) with a maximum propagation delay of 10 ns, programmed according to the equations in Figure 11-7, can be used to generate $\overline{CS}$. For a 25 MHz system, t_{CLK} low to $\overline{CS}$ low is less than or equal to 10 ns when this design is used. Should worst case conditions cause t_{CLK} low to $\overline{AS}$ low to exceed requirement (1), one wait state is inserted in the access to the FPCP; no other adverse effect occurs. Figure 11-8 shows the bus cycle timing for this interface. Refer to *MC68881/MC68882 Floating-Point Coprocessor User's Manual* for FPCP specifications.

The circuit that generates $\overline{CS}$ must meet another requirement. When a nonfloating-point access immediately follows a floating-point access, $\overline{CS}$ (for the floating-point access) must

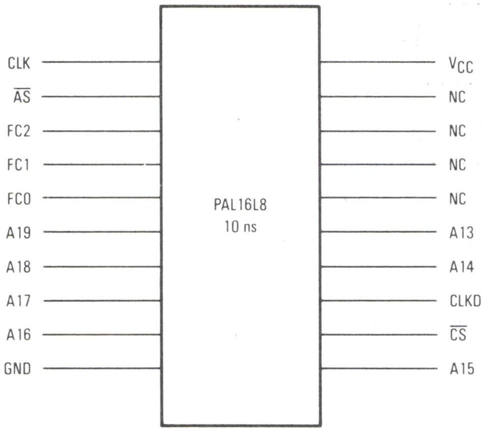


Figure 11-6. Chip-Select Generation PAL

```
PAL16l8
FPCP CS GENERATION CIRCUITRY FOR 25 MHz OPERATION
MOTOROLA INC., AUSTIN, TEXAS
CLK    AS     FC2    FC1    FC0    A19    A18    A17    A16    GND
A15    /CS    /CLKD  A14    A13    NC     NC     NC     NC     VCC

CS    = FC2    * FC1    * FC0             ;cpu space = $7
      * /A19   * /A18   * A17    * /A16   ;coprocessor access = $2
      * /A15   * /A14   * A13             ;coprocessor id = $1
      * /CLK                              ;qualified by MPU clock low

      + FC2    * FC1    * FC0             ;cpu space = $7
      * /A19   * /A18   * A17    * /A16   ;coprocessor access = $2
      * /A15   * /A14   * A13             ;coprocessor id = $1
      * /AS                               ;qualified by address strobe low

      + FC2    * FC1    * FC0
      * /A19   * /A18   * A17    * /A16   ;coprocessor access = $2
      * /A15   * /A14   * A13             ;coprocessor id = $1
      * /CLKD                             ;qualified by CLKD (delayed CLK)

CLKD = CLK
```

Description: There are three terms to the CS generation. The first term denotes the earliest time CS can be asserted. The second term is used to assert CS until the end of the FPCP access. The third term is to ensure that no race condition occurs in case of a late AS.

Figure 11-7. PAL Equations

be negated before $\overline{AS}$ and $\overline{DS}$ (for the subsequent access) are asserted. The PAL circuit previously described also meets this requirement.

For example, if a system has only one coprocessor, the full decoding of the ten signals (FC2–FC0 and A19–A13) provided by the PAL equations in Figure 11-7 is not absolutely

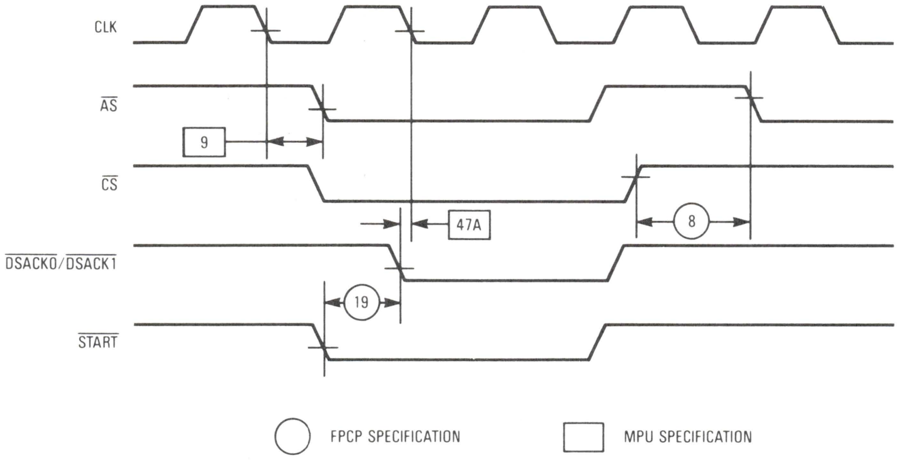

CLK
AS
9
CS
47A
8
DSACK0/DSACK1
19
START
FPCP SPECIFICATION
MPU SPECIFICATION

Figure 11-8. Bus Cycle Timing Diagram

necessary. It may be sufficient to use only FC1–FC0 and A17–A16. FC1–FC0 indicate when a bus cycle is operating in either CPU space ($7) or user-defined space ($3), and A17–A16 encode CPU space type as coprocessor space ($2). A15–A13 can be ignored in this case because they encode the coprocessor identification code (Cp-ID) used to differentiate between multiple coprocessors in a system. Motorola assemblers always default to a Cp-ID of $1 for floating-point instructions; this can be controlled with assembler directives if a different Cp-ID is desired or if multiple coprocessors exist in the system.

SECTION 12
ELECTRICAL CHARACTERISTICS

This section contains electrical specifications and associated timing information for the MC68020.

12.1 MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	−0.3 to +7.0	V
Input Voltage	V_{in}	−0.5 to +7.0	V
Operating Temperature Range	T_A	0 to 70	°C
Storage Temperature Range	T_{stg}	−55 to 150	°C

This device contains protective circuitry against damage due to high static voltages or electrical fields; however, it is advised that normal precautions be taken to avoid application of any voltages higher than maximum-rated voltages to this high-impedance circuit. Reliability of operation is enhanced if unused inputs are tied to an appropriate logic voltage level (e.g., either GND or V_{CC}).

12.2 THERMAL CHARACTERISTICS — PGA PACKAGE

Characteristic	Symbol	Value	Rating
Thermal Resistance — Ceramic			°C/W
Junction to Ambient	θ_{JA}	30*	
Junction to Case	θ_{JC}	11*	

*Estimated

12.3 POWER CONSIDERATIONS

The average chip-junction temperature, T_J, in °C can be obtained from:

$$T_J = T_A + (P_D \bullet \theta_{JA}) \tag{1}$$

where:

T_A = Ambient Temperature, °C
θ_{JA} = Package Thermal Resistance, Junction-to-Ambient, °C/W
P_D = $P_{INT} + P_{I/O}$
P_{INT} = $I_{CC} \times V_{CC}$, Watts — Chip Internal Power
$P_{I/O}$ = Power Dissipation on Input and Output Pins — User Determined

For most applications $P_{I/O} < P_{INT}$ and can be neglected.

The following is an approximate relationship between P_D and T_J (if $P_{I/O}$ is neglected):

$$P_D = K \div (T_J + 273°C) \tag{2}$$

Solving equations (1) and (2) for K gives:

$$K = P_D \bullet (T_A + 273°C) + \theta_{JA} \bullet P_D^2 \qquad (3)$$

where K is a constant pertaining to the particular part. K can be determined from equation (3) by measuring P_D (at equilibrium) for a known T_A. Using this value of K, the values of P_D and T_J can be obtained by solving equations (1) and (2) iteratively for any value of T_A.

The total thermal resistance of a package (θ_{JA}) can be separated into two components, θ_{JC} and θ_{CA}, representing the barrier to heat flow from the semiconductor junction to the package (case) surface (θ_{JC}) and from the case to the outside ambient (θ_{CA}). These terms are related by the equation:

$$\theta_{JA} = \theta_{JC} + \theta_{CA} \qquad (4)$$

θ_{JC} is device related and cannot be influenced by the user. However, θ_{CA} is user dependent and can be minimized by such thermal management techniques as heat sinks, ambient air cooling, and thermal convention. Thus, good thermal management on the part of the user can significantly reduce θ_{CA} so that θ_{JA} approximately equals θ_{JC}. Substitution of θ_{JC} for θ_{JA} in equation (1) will result in a lower semiconductor junction temperature.

Values for thermal resistance presented in this document, unless estimated, were derived using the procedure described in Motorola Reliability Report 7843, "Thermal Resistance Measurement Method for MC68XX Microcomponent Devices," and are provided for design purposes only. Thermal measurements are complex and dependent on procedure and setup. User derived values for thermal resistance may differ.

12.4 DC ELECTRICAL CHARACTERISTICS

($V_{CC} = 5.0$ Vdc ± 5%; GND = 0 Vdc; $T_A = 0$ to 70°C)

Characteristic		Symbol	Min	Max	Unit
Input High Voltage		V_{IH}	2.0	V_{CC}	V
Input Low Voltage		V_{IL}	GND −0.5	0.8	V
Input Leakage Current GND ≤ V_{in} ≤ V_{CC}	$\overline{\text{BERR}}$, $\overline{\text{BR}}$, $\overline{\text{BGACK}}$, CLK, $\overline{\text{IPL0}}$-$\overline{\text{IPL2}}$, $\overline{\text{AVEC}}$, $\overline{\text{CDIS}}$, $\overline{\text{DSACK0}}$, $\overline{\text{DSACK1}}$	I_{in}	−1	1.0	μA
	$\overline{\text{HALT}}$, $\overline{\text{RESET}}$		−20	20	
Hi-Z (Off-State) Leakage Current @ 2.4 V/0.5 V	A31–A0, $\overline{\text{AS}}$, $\overline{\text{DBEN}}$, $\overline{\text{DS}}$, D31–D0, FC2–FC0, R/$\overline{\text{W}}$, $\overline{\text{RMC}}$, SIZ1–SIZ0	I_{TSI}	−20	20	μA
Output High Voltage $I_{OH} = 400$ μA	A31–A0, $\overline{\text{AS}}$, $\overline{\text{BG}}$, D31–D0, $\overline{\text{DBEN}}$, $\overline{\text{DS}}$, $\overline{\text{ECS}}$, R/$\overline{\text{W}}$, $\overline{\text{IPEND}}$, $\overline{\text{OCS}}$, $\overline{\text{RMC}}$, SIZ1–SIZ0, FC2–FC0,	V_{OH}	2.4	—	V
Output Low Voltage		V_{OL}			V
$I_{OL} = 3.2$ mA	A31–A0, FC2–FC0, SIZ1–SIZ0, $\overline{\text{BG}}$, D31–D0		—	0.5	
$I_{OL} = 5.3$ mA	$\overline{\text{AS}}$, $\overline{\text{DS}}$, R/$\overline{\text{W}}$, $\overline{\text{RMC}}$, $\overline{\text{DBEN}}$, $\overline{\text{IPEND}}$		—	0.5	
$I_{OL} = 2.0$ mA	$\overline{\text{ECS}}$, $\overline{\text{OCS}}$		—	0.5	
$I_{OL} = 10.7$ mA	$\overline{\text{HALT}}$, $\overline{\text{RESET}}$		—	0.5	
Power Dissipation ($T_A = 0$°C)		P_D	—	2.0	W
Capacitance (see Note) $V_{in} = 0$ V, $T_A = 25$°C, f = 1 MHz		C_{in}	—	20	pF
Load Capacitance	$\overline{\text{ECS}}$, $\overline{\text{OCS}}$ All Other	C_L	—	50 130	pF

NOTE: Capacitance is periodically sampled rather than 100% tested.

12.5 AC ELECTRICAL SPECIFICATIONS — CLOCK INPUT (see Figure 12-1)

Num.	Characteristic	12.5 MHz		16.67 MHz		20 MHz		25 MHz*		33.33 MHz		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	
	Frequency of Operation	8	12.5	8	16.67	12.5	20	12.5	25	12.5	33.33	MHz
1	Cycle Time	80	125	60	125	50	80	40	80	30	80	ns
2, 3	Clock Pulse Width (Measured from 1.5 V to 1.5 V for 25 and 33.33 MHz)	32	87	24	95	20	54	19	61	14	66	ns
4, 5	Rise and Fall Times	—	5	—	5	—	5	—	4	—	3	ns

*These specifications represent an improvement over previously published specifications for the 25 MHz MC68020 and are valid only for product bearing date codes of 8827 and later.

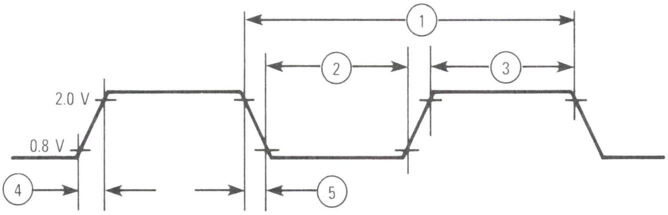


Figure 12-1. Clock Input Timing Diagram

12.6 AC ELECTRICAL SPECIFICATIONS — READ AND WRITE CYCLES

(V_{CC} = 5.0 Vdc ± 5%; GND = 0 Vdc; T_A = 0 to 70°C; see Figures 12-2, 12-3, and 12-4)

Num.	Characteristic	12.5 MHz		16.67 MHz		20 MHz		25 MHz*		33.33 MHz		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	
6	Clock High to Address, FC, Size, RMC Valid	0	40	0	30	0	25	0	25	0	21	ns
6A	Clock High to ECS, OCS Asserted	0	30	0	20	0	15	0	12	0	10	ns
7	Clock High to Address, Data, FC, Size, RMC, High Impedance	0	80	0	60	0	50	0	40	0	30	ns
8	Clock High to Address, FC, Size, RMC Invalid	0	—	0	—	0	—	0	—	0	—	ns
9	Clock Low to AS, DS Asserted	3	40	3	30	3	25	3	18	3	15	ns
9A[1]	AS to DS Assertion (Read) (Skew)	−20	20	−15	15	−10	10	−10	10	−10	10	ns
9B[11]	AS Asserted to DS Asserted (Write)	47	—	37	—	32	—	27	—	22	—	ns
10	ECS Width Asserted	25	—	20	—	15	—	15	—	10	—	ns
10A	OCS Width Asserted	25	—	20	—	15	—	15	—	10	—	ns
10B[7]	ECS, OCS Width Negated	20	—	15	—	10	—	5	—	5	—	ns
11	Address, FC, Size, RMC Valid to AS (and DS Asserted Read)	20	—	15	—	10	—	6	—	5	—	ns
12	Clock Low to AS, DS Negated	0	40	0	30	0	25	0	15	0	15	ns
12A	Clock Low to ECS, OCS Negated	0	40	0	30	0	25	0	15	0	15	ns
13	AS, DS Negated to Address, FC, Size RMC Invalid	20	—	15	—	10	—	10	—	5	—	ns
14	AS (and DS Read) Width Asserted	120	—	100	—	85	—	70	—	50	—	ns
14A	DS Width Asserted Write	50	—	40	—	38	—	30	—	25	—	ns
15	AS, DS Width Negated	50	—	40	—	38	—	30	—	23	—	ns
15A[8]	DS Negated to AS Asserted	45	—	35	—	30	—	25	—	18	—	ns
16	Clock High to AS, DS, R/W, DBEN High Impedance	—	80	—	60	—	50	—	40	—	30	ns
17	AS, DS Negated to R/W Invalid	20	—	15	—	10	—	10	—	5	—	ns
18	Clock High to R/W High	0	40	0	30	0	25	0	20	0	15	ns
20	Clock High to R/W Low	0	40	0	30	0	25	0	20	0	15	ns
21	R/W High to AS Asserted	20	—	15	—	10	—	5	—	5	—	ns
22	R/W Low to DS Asserted (Write)	90	—	75	—	60	—	50	—	35	—	ns
23	Clock High to Data Out Valid	—	40	—	30	—	25	—	25	—	18	ns
25	DS Negated to Data Out Invalid	20	—	15	—	10	—	5	—	5	—	ns
25A[9]	DS Negated to DBEN Negated (Write)	20	—	15	—	10	—	5	—	5	—	ns
26	Data Out Valid to DS Asserted (Write)	20	—	15	—	10	—	5	—	5	—	ns
27	Data-In Valid to Clock Low (Data Setup)	10	—	5	—	5	—	5	—	5	—	ns
27A	Late BERR/HALT Asserted to Clock Low Setup Time	25	—	20	—	15	—	10	—	5	—	ns
28	AS, DS Negated to DSACKx, BERR, HALT, AVEC Negated	0	110	0	80	0	65	0	50	0	40	ns

— Continued —

12.6 AC ELECTRICAL SPECIFICATIONS (Continued)

Num.	Characteristic	12.5 MHz		16.67 MHz		20 MHz		25 MHz*		33.33 MHz		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	
29	$\overline{\mathrm{DS}}$ Negated to Data-In Invalid (Data-In Hold Time)	0	—	0	—	0	—	0	—	0	—	ns
29A	$\overline{\mathrm{DS}}$ Negated to Data-In (High Impedance)	—	80	—	60	—	50	—	40	—	30	ns
31[2]	$\overline{\mathrm{DSACKx}}$ Asserted to Data-In Valid	—	60	—	50	—	43	—	32	—	17	ns
31A[3]	$\overline{\mathrm{DSACKx}}$ Asserted to $\overline{\mathrm{DSACKx}}$ Valid ($\overline{\mathrm{DSACK}}$ Asserted Skew)	—	20	—	15	—	10	—	10	—	10	ns
32	$\overline{\mathrm{RESET}}$ Input Transition Time	—	1.5	—	1.5	—	1.5	—	1.5	—	1.5	Clks
33	Clock Low to $\overline{\mathrm{BG}}$ Asserted	0	40	0	30	0	25	0	20	0	20	ns
34	Clock Low to $\overline{\mathrm{BG}}$ Negated	0	40	0	30	0	25	0	20	0	20	ns
35	$\overline{\mathrm{BR}}$ Asserted to $\overline{\mathrm{BG}}$ Asserted ($\overline{\mathrm{RMC}}$ Not Asserted)	1.5	3.5	1.5	3.5	1.5	3.5	1.5	3.5	1.5	3.5	Clks
37	$\overline{\mathrm{BGACK}}$ Asserted to $\overline{\mathrm{BG}}$ Negated	1.5	3.5	1.5	3.5	1.5	3.5	1.5	3.5	1.5	3.5	Clks
37A[6]	$\overline{\mathrm{BGACK}}$ Asserted to $\overline{\mathrm{BR}}$ Negated	0	1.5	0	1.5	0	1.5	0	1.5	0	1.5	Clks
39	$\overline{\mathrm{BG}}$ Width Negated	120	—	90	—	75	—	60	—	50	—	ns
39A	$\overline{\mathrm{BG}}$ Width Asserted	120	—	90	—	75	—	60	—	50	—	ns
40	Clock High to $\overline{\mathrm{DBEN}}$ Asserted (Read)	0	40	0	30	0	25	0	20	0	15	ns
41	Clock Low to $\overline{\mathrm{DBEN}}$ Negated (Read)	0	40	0	30	0	25	0	20	0	15	ns
42	Clock Low to $\overline{\mathrm{DBEN}}$ Asserted (Write)	0	40	0	30	0	25	0	20	0	15	ns
43	Clock High to $\overline{\mathrm{DBEN}}$ Negated (Write)	0	40	0	30	0	25	0	20	0	15	ns
44	R/$\overline{\mathrm{W}}$ Low to $\overline{\mathrm{DBEN}}$ Asserted (Write)	20	—	15	—	10	—	10	—	5	—	ns
45[5]	$\overline{\mathrm{DBEN}}$ Width Asserted Read Write	80 160	— —	60 120	— —	50 100	— —	40 80	— —	30 60	— —	ns
46	R/$\overline{\mathrm{W}}$ Width Valid (Write or Read)	180	—	150	—	125	—	100	—	75	—	ns
47A	Asynchronous Input Setup Time	10	—	5	—	5	—	5	—	5	—	ns
47B	Asynchronous Input Hold Time	20	—	15	—	15	—	10	—	10	—	ns
48[4]	$\overline{\mathrm{DSACKx}}$ Asserted to $\overline{\mathrm{BERR}}$, $\overline{\mathrm{HALT}}$ Asserted	—	35	—	30	—	20	—	18	—	15	ns
53	Data Out Hold from Clock High	0	—	0	—	0	—	0	—	0	—	ns
55	R/$\overline{\mathrm{W}}$ Valid to Data Bus Impedance Change	40	—	30	—	25	—	20	—	20	—	ns
56	$\overline{\mathrm{RESET}}$ Pulse Width (Reset Instruction)	512	—	512	—	512	—	512	—	512	—	Clks
57	$\overline{\mathrm{BERR}}$ Negated to $\overline{\mathrm{HALT}}$ Negated (Rerun)	0	—	0	—	0	—	0	—	0	—	ns
58[10]	$\overline{\mathrm{BGACK}}$ Negated to Bus Driven	1	—	1	—	1	—	1	—	1	—	Clks
59[10]	$\overline{\mathrm{BG}}$ Negated to Bus Driven	1	—	1	—	1	—	1	—	1	—	Clks

*These specifications represent an improvement over previously published specifications for the 25 MHz MC68020 and are valid only for product bearing date codes of 8827 and later.

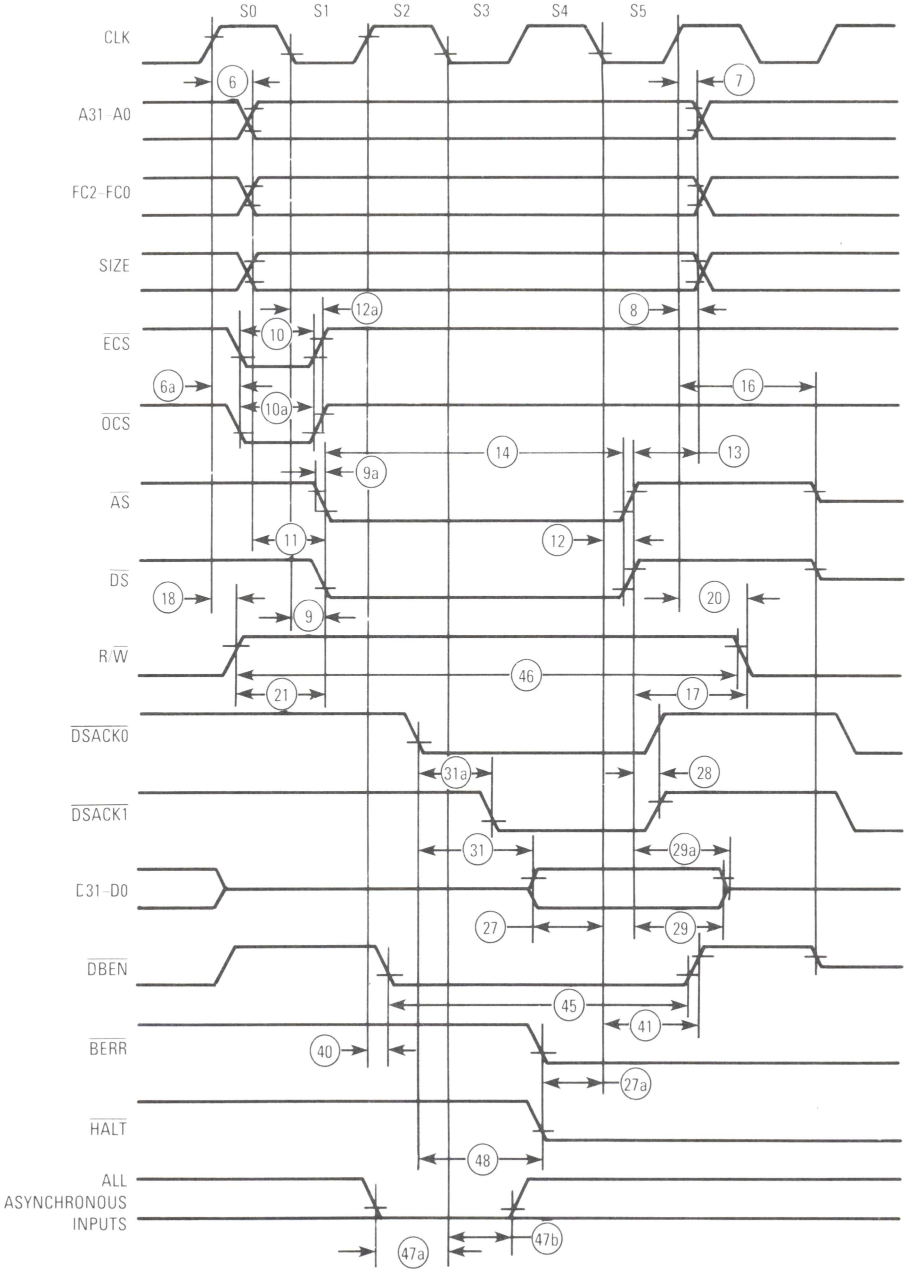
— Continued —

12.6 AC ELECTRICAL SPECIFICATIONS (Concluded)

NOTES:

1. This number can be reduced to 5 nanoseconds if strobes have equal loads.
2. If the asynchronous setup time (#47A) requirements are satisfied, the DSACKx low to data setup time (#31) and DSACKx low to BERR low setup time (#48) can be ignored. The data must only satisfy the data-in to clock low setup time (#27) for the following clock cycle and BERR must only satisfy the late BERR low to clock low setup time (#27A) for the following clock cycle.
3. This parameter specifies the maximum allowable skew between DSACK0 to DSACK1 asserted or DSACK1 to DSACK0 asserted; specification #47A must be met by DSACK0 or DSACK1.
4. This specification applies to the first (DSACK0 or DSACK1) DSACKx signal asserted. In the absence of DSACKx, BERR is an asynchronous input using the asynchronous input setup time (#47A).
5. DBEN may stay asserted on consecutive write cycles.
6. The minimum values must be met to guarantee proper operation. If this maximum value is exceeded, BG may be reasserted.
7. This specification indicates the minimum high time for ECS and OCS in the event of an internal cache hit followed immediately by a cache miss or operand cycle.
8. This specification guarantees operation with the MC68881 MC68882, which specifies a minimum time for DS negated to AS asserted (specification #13A in the *MC68881 MC68882 User's Manual*). Without this specification, incorrect interpretation of specifications #9A and #15 would indicate that the MC68020 does not meet the MC68881 MC68882 requirements.
9. This specification allows a system designer to guarantee data hold times on the output side of data buffers that have output enable signals generated with DBEN.
10. These specifications allow system designers to guarantee that an alternate bus master has stopped driving the bus when the MC68020 regains control of the bus after an arbitration sequence.
11. This specification allows system designers to qualify the CS signal of an MC68881 MC68882 with AS (allowing 7ns for a gate delay and still meet the CS to DS setup time requirement (specification 8B) of the MC68881 MC68882.

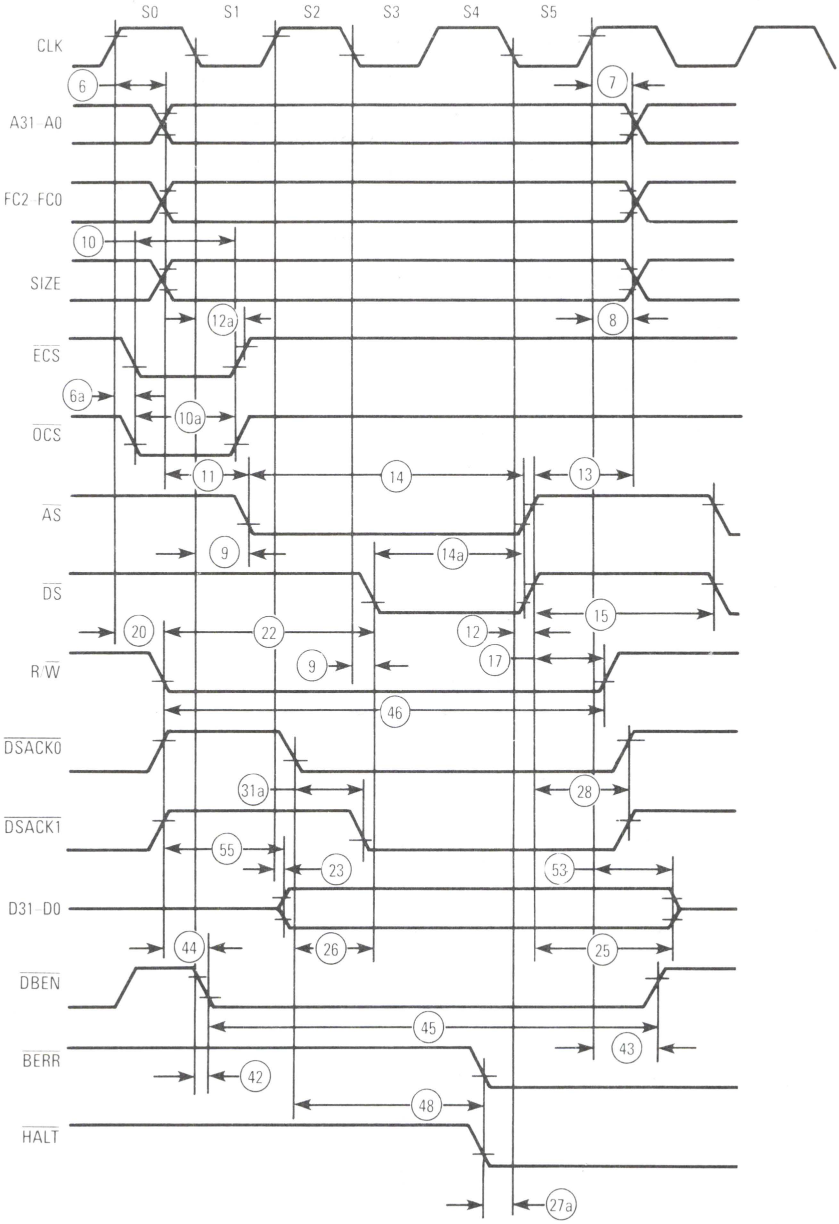
These waveforms should only be referenced in regard to the edge-to-edge measurement of the timing specifications. They are not intended as a functional description of the input and output signals. Refer to other functional descriptions and their related diagrams for device operation.


S0
S1
S2
S3
S4
S5
CLK
A31–A0
FC2–FC0
SIZE
ECS
OCS
AS
DS
R/W
DSACK0
DSACK1
D31–D0
DBEN
BERR
HALT
ALL
ASYNCHRONOUS
INPUTS
6
7
6a
8
9
9a
10
10a
11
12
12a
13
14
16
17
18
20
21
27
27a
28
29
29a
31
31a
40
41
45
46
47a
47b
48

NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside, and pass through the range such that the rise or fall will be linear between 0.8 volts and 2.0 volts.

Figue 12-2. Read-Cycle Timing Diagram

These waveforms should only be referenced in regard to the edge-to-edge measurement of the timing specifications. They are not intended as a functional description of the input and output signals. Refer to other functional descriptions and their related diagrams for device operation.

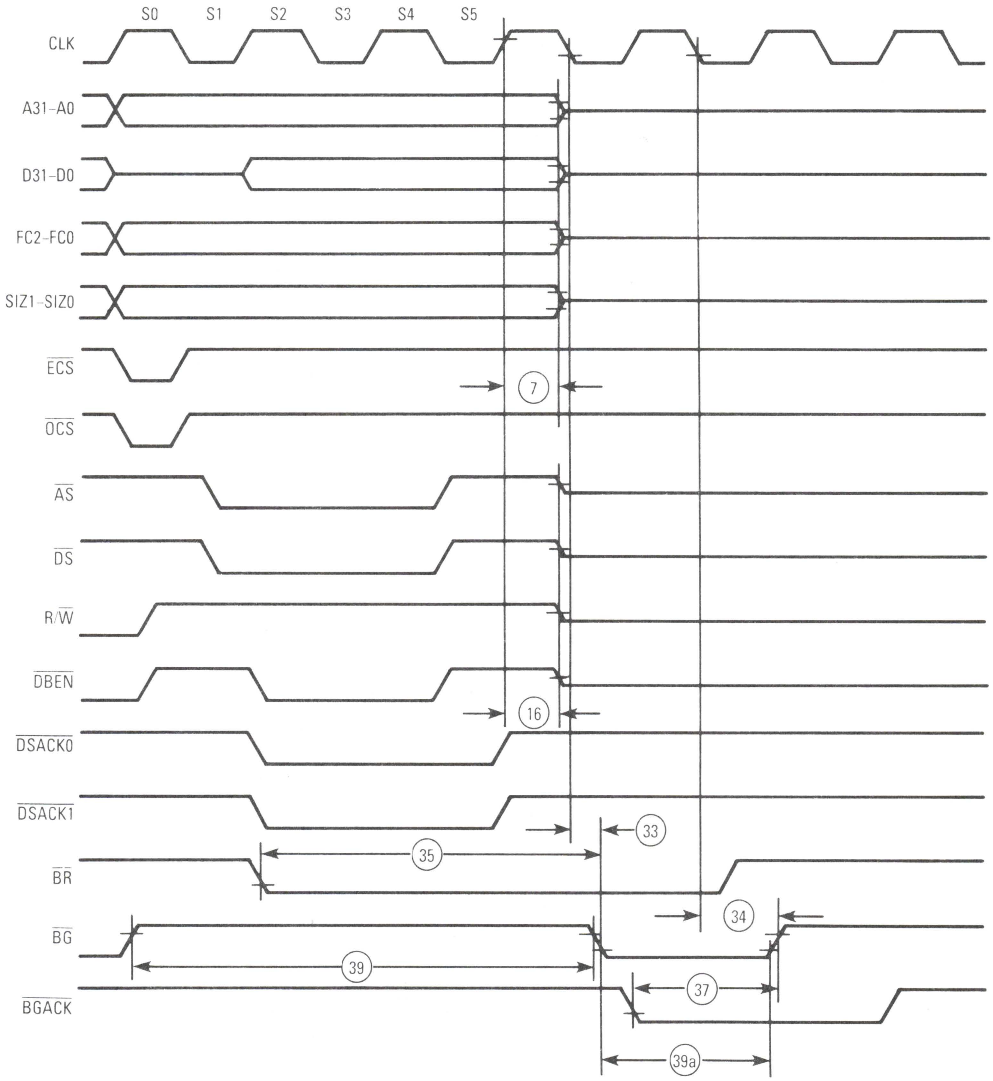


S0
S1
S2
S3
S4
S5
CLK
6
7
A31–A0
FC2–FC0
10
SIZE
12a
8
ECS
6a
10a
OCS
11
14
13
AS
9
14a
DS
15
20
22
12
9
17
R/W
46
DSACK0
31a
28
DSACK1
55
23
53
D31–D0
44
26
25
DBEN
45
43
BERR
42
48
HALT
27a

NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside, and pass through the range such that the rise or fall will be linear between 0.8 volts and 2.0 volts.

Figure 12-3. Write-Cycle Timing Diagram

These waveforms should only be referenced in regard to the edge-to-edge measurement of the timing specifications. They are not intended as a functional description of the input and output signals. Refer to other functional descriptions and their related diagrams for device operation.


S0
S1
S2
S3
S4
S5
CLK
A31–A0
D31–D0
FC2–FC0
SIZ1–SIZ0
ECS
OCS
AS
DS
R/W
DBEN
DSACK0
DSACK1
BR
BG
BGACK
7
16
33
35
34
39
37
39a

NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside, and pass through the range such that the rise or fall will be linear between 0.8 volts and 2.0 volts.

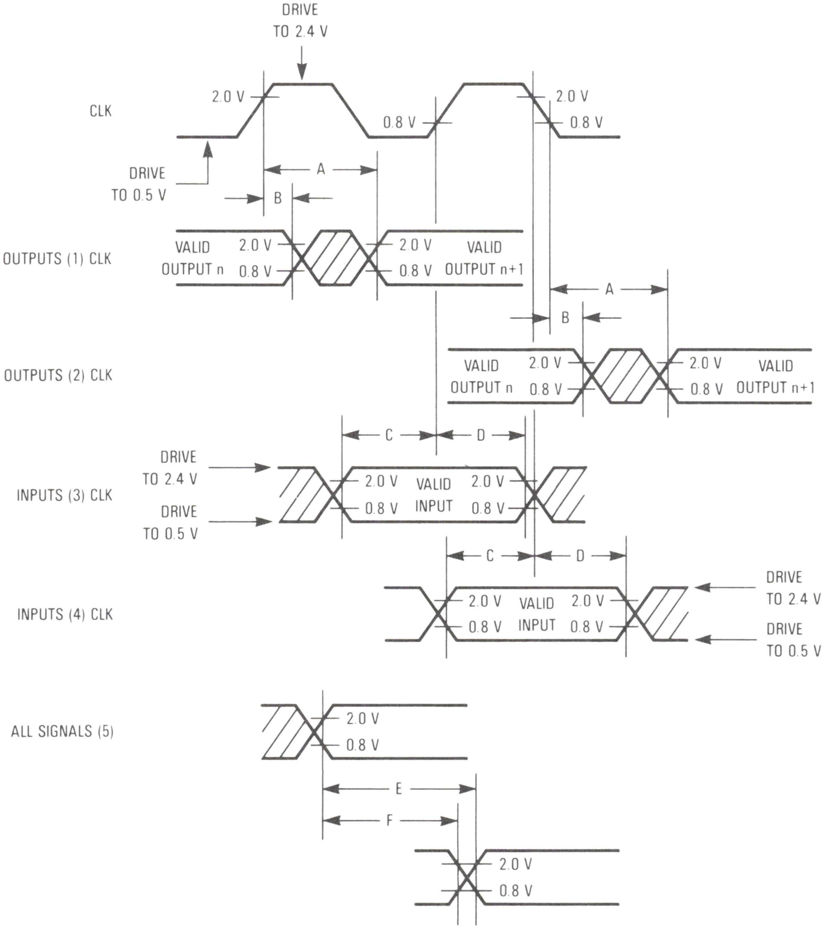
Figure 12-4. Bus Arbitration Timing Diagram

12.7 AC ELECTRICAL SPECIFICATIONS DEFINITIONS

The AC specifications presented consist of output delays, input setup and hold times, and signal skew times. All signals are specified relative to an appropriate edge of the MC68020 clock input and, possibly, relative to one or more other signals.

The measurement of the AC specifications is defined by the waveforms in Figure 12-5. To test the parameters guaranteed by Motorola, inputs must be driven to the voltage levels specified in Figure 12-5. Outputs of the MC68020 are specified with minimum and/or maximum limits, as appropriate, and are measured as shown. Inputs to the MC68020 are specified with minimum and, as appropriate, maximum setup and hold times, and are measured as shown. Finally, the measurements for signal-to-signal specifications are also shown.

Note that the testing levels used to verify conformance of the MC68020 to the AC specifications does not affect the guaranteed DC operation of the device as specified in the DC electrical characteristics.


DRIVE
TO 2.4 V
CLK
2.0 V
0.8 V
DRIVE
TO 0.5 V
A
B
OUTPUTS (1) CLK
VALID
OUTPUT n
VALID
OUTPUT n+1
OUTPUTS (2) CLK
C
D
INPUTS (3) CLK
VALID
INPUT
INPUTS (4) CLK
ALL SIGNALS (5)
E
F

NOTES:

1 - This output timing is applicable to all parameters specified relative to the rising edge of the clock
2 - This output timing is applicable to all parameters specified relative to the falling edge of the clock
3 - This input timing is applicable to all parameters specified relative to the rising edge of the clock
4 - This input timing is applicable to all parameters specified relative to the falling edge of the clock
5 - This timing is applicable to all parameters specified relative to the assertion/negation of another signal

LEGEND:

A - Maximum output delay specification
B - Minimum output hold time
C - Minimum input setup time specification
D - Minimum input hold time specification
E - Signal valid to signal valid specification (maximum or minimum)
F - Signal valid to signal invalid specification (maximum or minimum)

Figure 12-5. Drive Levels and Test Points for AC Specifications

SECTION 13
ORDERING INFORMATION AND MECHNICAL DATA

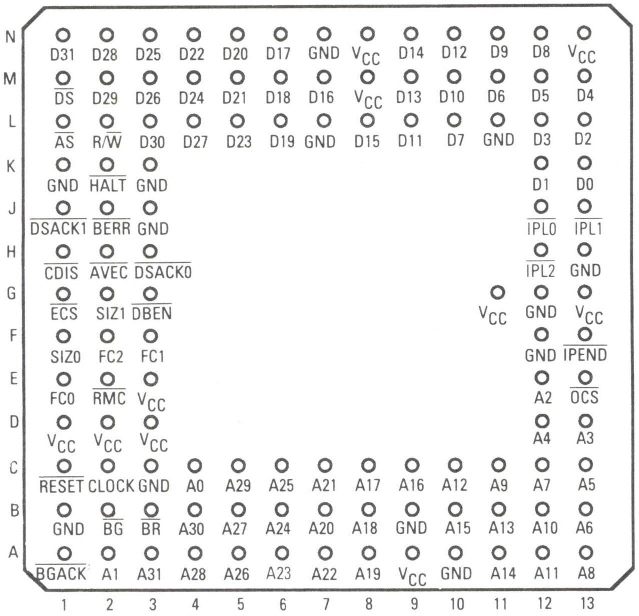
This section contains the pin assignments and package dimensions of the MC68020. In addition, detailed information is provided to be used as a guide when ordering.

13.1 STANDARD MC68020 ORDERING INFORMATION

Package Type	Frequency (MHz)	Temperature	Order Number
Pin Grid Array	12.5	0°C to 70°C	MC68020RC12
RC Suffix	16.7	0°C to 70°C	MC68020RC16
	20.0	0°C to 70°C	MC68020RC20
	25.0	0°C to 70°C	MC68020RC25
	33.33	0°C to 70°C	MC68020RC33
RL Suffix	16.7	0°C to 70°C	MC68020RL16
	20.0	0°C to 70°C	MC68020RL20
	25.0	0°C to 70°C	MC68020RL25
	33.33	0°C to 70°C	MC68020RL33
Ceramic Surface	16.7	0°C to 70°C	MC68020FE16
Mount	20.0	0°C to 70°C	MC68020FE20
FE Suffix	25.0	0°C to 70°C	MC68020FE25

13.2 PIN ASSIGNEMENTS AND PACKAGE DIMENSIONS

13.2.1 RC, RL Suffix — Pin Assignment

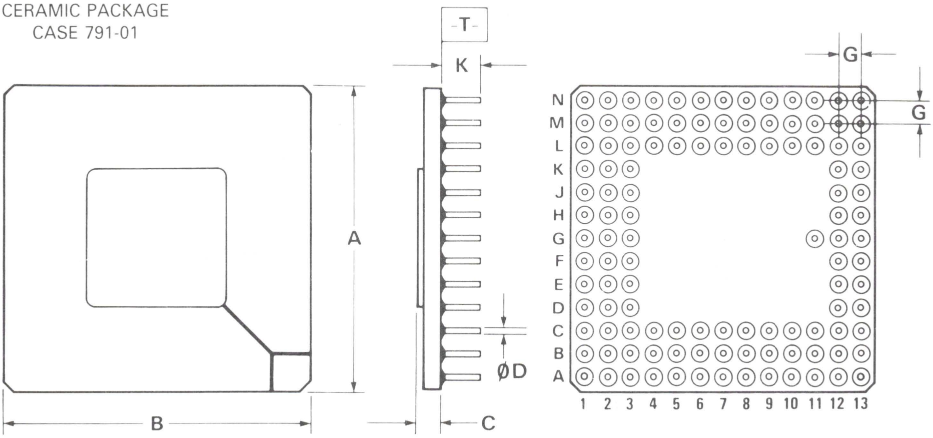

N D31 D28 D25 D22 D20 D17 GND VCC D14 D12 D9 D8 VCC
M DS D29 D26 D24 D21 D18 D16 VCC D13 D10 D6 D5 D4
L AS R/W D30 D27 D23 D19 GND D15 D11 D7 GND D3 D2
K GND HALT GND D1 D0
J DSACK1 BERR GND IPL0 IPL1
H CDIS AVEC DSACK0 IPL2 GND
G ECS SIZ1 DBEN VCC GND VCC
F SIZ0 FC2 FC1 GND IPEND
E FC0 RMC VCC A2 OCS
D VCC VCC VCC A4 A3
C RESET CLOCK GND A0 A29 A25 A21 A17 A16 A12 A9 A7 A5
B GND BG BR A30 A27 A24 A20 A18 GND A15 A13 A10 A6
A BGACK A1 A31 A28 A26 A23 A22 A19 VCC GND A14 A11 A8
1 2 3 4 5 6 7 8 9 10 11 12 13

The V_{CC} and GND pins are separated into three groups to provide individual power supply connections for the address bus buffers, data bus buffers, and all other output buffers and internal logic.

Group	V_{CC}	GND
Address Bus	A9, D3	A10, B9, C3, F12
Data Bus	M8, N8, N13	L7, L11, N7, K3
Logic	D1, D2, E3, G11, G13	G12, H13, J3, K1
Clock		B1

13.2.2 RC Suffix — Package Dimensions

RC SUFFIX
CERAMIC PACKAGE
CASE 791-01



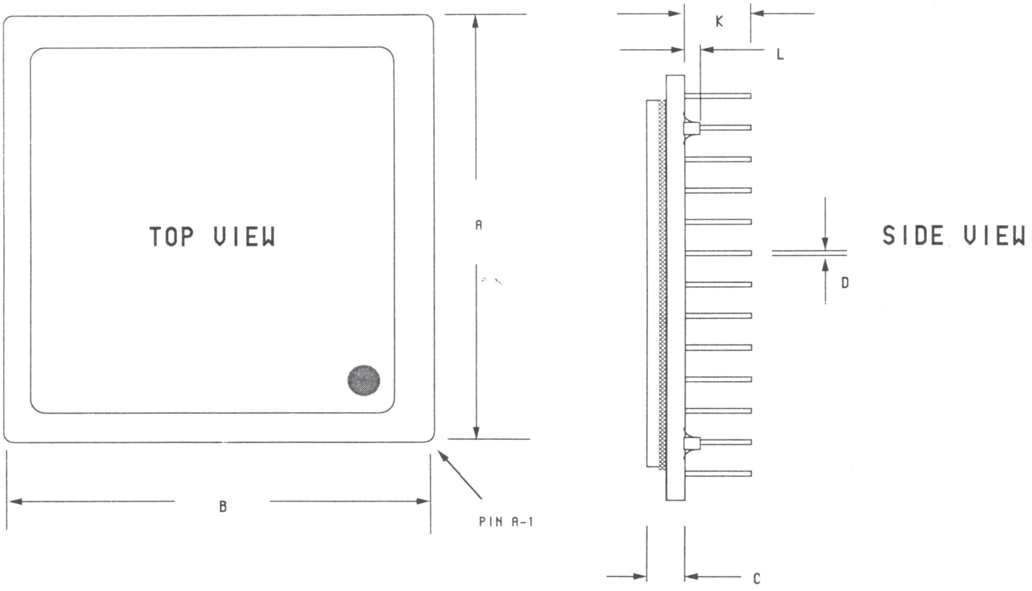
NOTES:

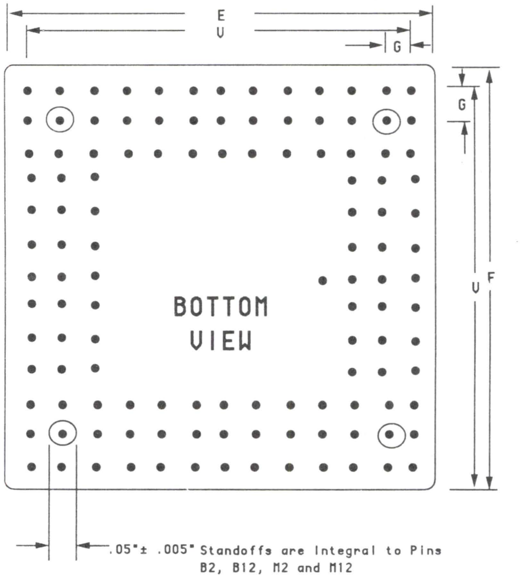
1. A AND B ARE DATUMS AND T IS A DATUM SURFACE.
2. POSITIONAL TOLERANCE FOR LEADS (114 PLACES).
   ⌖ 0.13 (0.005) Ⓜ | T | A Ⓢ | B Ⓢ
3. DIMENSIONING AND TOLERANCING PER Y14.5M, 1982.
4. CONTROLLING DIMENSION: INCH

	MILLIMETERS		INCHES	
DIM	**MIN**	**MAX**	**MIN**	**MAX**
A	34.04	35.05	1.340	1.380
B	34.04	35.05	1.340	1.380
C	2.54	3.81	0.100	0.150
D	0.43	0.55	0.017	0.022
G	2.54 BSC		0.100 BSC	
K	4.32	4.95	0.170	0.195

13.2.3 RL Suffix — Package Dimensions

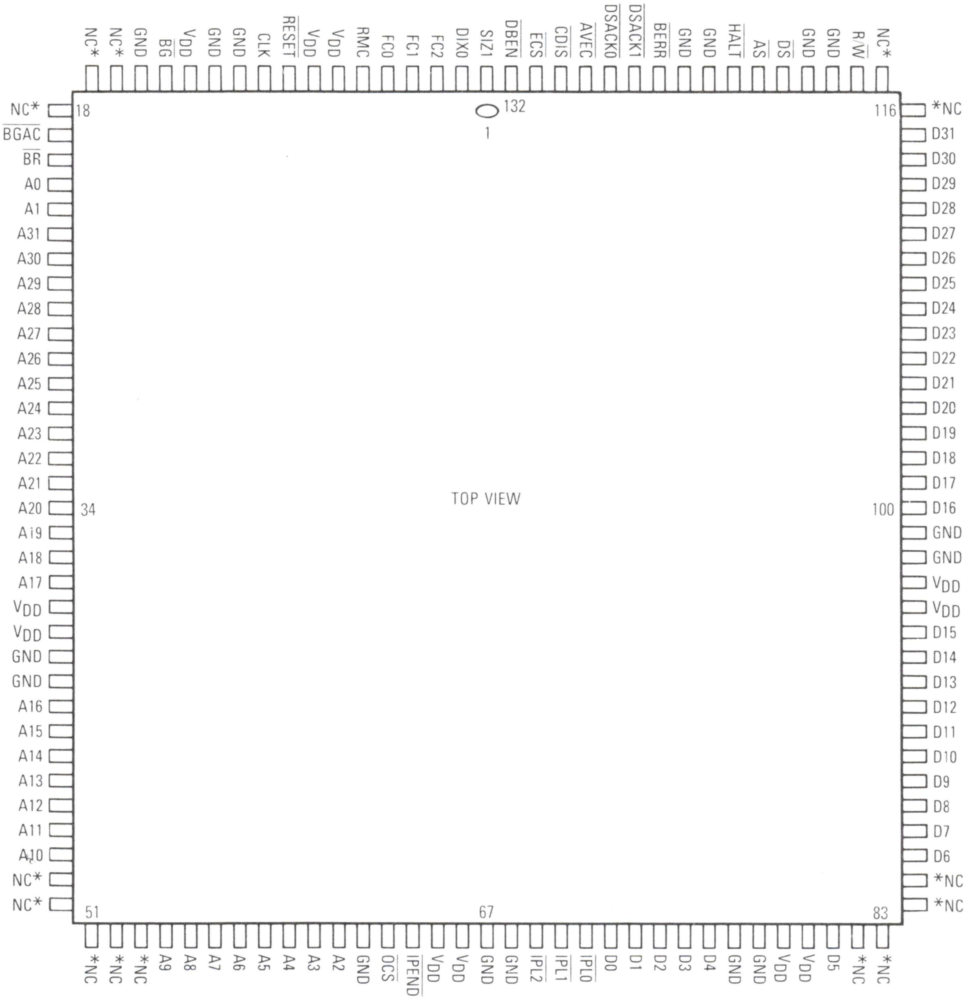
RL SUFFIX
(GLASS-SEAL CERAMIC PGA W TINNED LEADS)
PRELIMINARY MECHANICAL DATA


TOP VIEW
A
B
PIN A-1
K
L
SIDE VIEW
D
C


E
V
G
G
V
F
BOTTOM VIEW
.05"± .005" Standoffs are Integral to Pins B2, B12, M2 and M12

	MILLIMETERS		INCHES	
DIM	MIN	MAX	MIN	MAX
A	34.04	35.05	1.34	1.38
B	34.04	35.05	1.34	1.38
C	2.54	3.81	0.1	0.15
D	0.43	0.56	0.017	0.022
E	34.04	35.05	1.34	1.38
F	34.04	35.05	1.34	1.38
G	2.54 BSC		.100 BSC	
K	4.32	4.95	0.17	0.195
L	1.02	1.52	0.04	0.06
V	30.48 BSC		1.200 BSC	

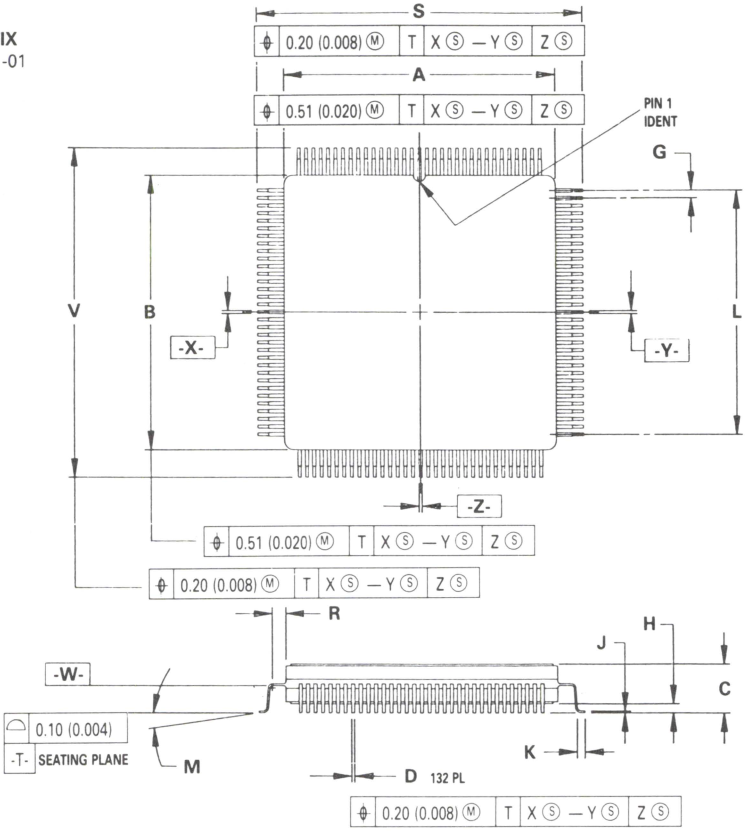
13.2.4 FE Suffix — Pin Assignment


NC*
NC*
GND
BG
VDD
GND
GND
CLK
RESET
VDD
VDD
RMC
FC0
FC1
FC2
DIX0
SIZ1
DBEN
ECS
CDIS
AVEC
DSACK0
DSACK1
BERR
GND
GND
HALT
AS
DS
GND
GND
R/W
NC*
NC*
BGAC
BR
A0
A1
A31
A30
A29
A28
A27
A26
A25
A24
A23
A22
A21
A20
A19
A18
A17
VDD
VDD
GND
GND
A16
A15
A14
A13
A12
A11
A10
NC*
NC*
*NC
D31
D30
D29
D28
D27
D26
D25
D24
D23
D22
D21
D20
D19
D18
D17
D16
GND
GND
VDD
VDD
D15
D14
D13
D12
D11
D10
D9
D8
D7
D6
*NC
*NC
*NC
*NC
*NC
A9
A8
A7
A6
A5
A4
A3
A2
GND
OCS
IPEND
VDD
VDD
GND
GND
IPL2
IPL1
IPL0
D0
D1
D2
D3
D4
GND
GND
VDD
VDD
D5
*NC
*NC
18
132
1
116
34
TOP VIEW
100
51
67
83

*NC — Do not connect to this pin.

13.2.5 FE Suffix — Package Dimensions

FE SUFFIX
CASE 831-01



S
⌖ 0.20 (0.008) Ⓜ T X Ⓢ —Y Ⓢ Z Ⓢ
A
⌖ 0.51 (0.020) Ⓜ T X Ⓢ —Y Ⓢ Z Ⓢ
PIN 1
IDENT
G
V
B
-X-
L
-Y-
-Z-
⌖ 0.51 (0.020) Ⓜ T X Ⓢ —Y Ⓢ Z Ⓢ
⌖ 0.20 (0.008) Ⓜ T X Ⓢ —Y Ⓢ Z Ⓢ
R
H
J
C
-W-
⌓ 0.10 (0.004)
-T- SEATING PLANE
M
K
D 132 PL
⌖ 0.20 (0.008) Ⓜ T X Ⓢ —Y Ⓢ Z Ⓢ

	MILLIMETERS		INCHES	
DIM	**MIN**	**MAX**	**MIN**	**MAX**
A	21.85	22.86	0.860	0.900
B	21.85	22.86	0.860	0.900
C	3.94	4.31	0.155	0.170
D	0.204	0.292	0.0080	0.0115
G	0.64 BSC		0.025 BSC	
H	0.64	0.88	0.025	0.035
J	0.13	0.20	0.005	0.008
K	0.51	0.76	0.020	0.030
L	20.32 REF		0.800 REF	
M	0°	8°	0°	8°
R	0.64	—	0.025	—
S	27.31	27.55	1.075	1.085
V	27.31	27.55	1.075	1.085

NOTES:

1. DIMENSIONING AND TOLERANCING PER ANSI Y14.5M, 1982.
2. CONTROLLING DIMENSION: INCH.
3. DIM A AND B DEFINE MAXIMUM CERAMIC BODY DIMENSIONS INCLUDING GLASS PROTRUSION AND MISMATCH OF CERAMIC BODY TOP AND BOTTOM.
4. DATUM PLANE -W- IS LOCATED AT THE UNDERSIDE OF LEADS WHERE LEADS EXIT PACKAGE BODY.
5. DATUMS X–Y AND Z TO BE DETERMINED WHERE CENTER LEADS EXIT PACKAGE BODY AT DATUM -W-.
6. DIM S AND V TO BE DETERMINED AT SEATING PLANE, DATUM -T-.
7. DIM A AND B TO BE DETERMINED AT DATUM PLANE -W-.

INDEX

— A —

- Absolute Long Address Mode, 2-16
- Absolute Short Address Mode, 2-16
- AC Electrical Specifications,
  - Clock Input, 12-3
  - Definitions, 12-6
  - General, 12-4
  - Read/Write Cycles, 12-4
- AC Electrical Timing Diagrams, 12-7
- Access Levels, 11-3
  - Module Call, 11-5
  - Module Return, 11-5
- Acknowledge, Breakpoint, 8-6
- Address Bus, 5-3, 7-3, 7-22ff
- Address Encoding, CPU Space, 7-38
- Address Error Exception, 8-6, 9-54
- Address Offset Encoding, 7-6
- Address Register
  - Direct Mode, 2-8
  - Indirect Displacement Mode, 2-9
  - Indirect Index (Base Displacement) Mode, 2-10
  - Indirect Index (8-Bit Displacement) Mode, 2-10
  - Indirect Mode, 2-8
  - Indirect Postincrement Mode, 2-8
  - Indirect Predecrement Mode, 2-9
- Address Registers, 1-3, 2-3
- Address Space Types, 4-4
- Address Strobe Signal, 5-4, 7-2, 7-3, 7-22ff
- Addressing,
  - Capabilities, 2-19
  - Compatibility, M68000, 2-27
  - Indexed, 2-19, 2-20
  - Indirect, 2-22, 2-23
  - Indirect Absolute Memory, 2-21
  - Mode Summary, 2-23
  - Modes, 1-5, 2-5, 2-19
  - Structure, 2-22
- Arbitration, Bus, 7-54
- Arithmetic/Logical Instruction,
  - Immediate, Timing Table, 10-30
  - Timing Table, 10-29
- AS Signal, 5-4, 7-2, 7-3, 7-22ff
- Assignment, Pin, 13-2
- Assignments, Exception Vector, 8-2
- Asynchronous
  - Bus Operation, 7-21
  - Byte
    - Read Cycle, 32-Bit Port, Timing, 7-25
    - Read Cycle Flowchart, 7-24
- Asynchronous
  - Byte (Continued)
    - Read-Modify-Write Cycle, 32-Bit Port, Timing, 7-36
    - Write Cycle, 32-Bit Port, Timing, 7-30
  - Cycle Signal Assertion Results, 7-47
  - Long Word Read Cycle Flowchart, 7-23
  - Read Cycle, 7-23
    - 32-Bit Port, Timing, 7-29
  - Read-Modify-Write Cycle, 7-34
    - Flowchart, 7-35
  - Sample Window, 7-2
  - Word
    - Read Cycle, 32-Bit Port, Timing, 7-25
    - Write Cycle, 32-Bit Port, Timing, 7-30
  - Write Cycle, 7-28
    - 32-Bit Port, Timing, 7-29
    - Flowchart, 7-28
- Autovector Interrupt Acknowledge Cycle, 7-39
  - Timing, 7-42
- Autovector Signal, 5-5, 7-5, 8-14
- AVEC Signal, 5-5, 7-5, 8-14
- A1–A0 Signals, 7-6, 7-7, 7-17ff
- A31–A0 Signals, 5-3, 7-3
- A7–A0, 1-3

— B —

- BERR Signal, 5-6, 7-5, 7-22ff, 8-4, 8-15, 8-18,
- BG Signal, 5-5, 7-55
- BGACK Signal, 5-6, 7-55
- Binary Coded Decimal Instruction Timing Table, 10-31
- Binary Coded Decimal Instructions, 3-8
- Bit,
  - C, 6-3
  - CA, 9-27
  - CE, 6-3
  - Clear Cache, 6-3
  - Clear Entry in Cache, 6-3
  - DR, 9-28
  - E, 6-3
  - Enable Cache, 6-3
  - F, 6-3
  - Freeze Cache, 6-3
  - PC, 9-28
- Bit Field
  - Instruction Timing Table, 10-35
  - Instructions, 3-7
  - Operations, 3-198

— B —

- Bit Manipulation
  - Instruction Timing Table, 10-34
  - Instructions, 3-7
- BKPT Instruction, 7-40, 8-14
- Block Diagram, 1-2
- BR Signal, 5-5, 7-34
- Branch on Coprocessor Condition Instruction, 9-10
- Breakpoint Acknowledge, 8-6
  - Cycle, 7-40
    - Exception Signaled, Timing, 7-45
    - Timing, 7-44
  - Flowchart, 7-43
- Breakpoint Instruction, 7-40, 8-14
  - Exception, 8-15
- Bus,
  - Address, 5-3, 7-3, 7-22ff
  - Arbitration, 7-54
    - Bus Inactive, Timing, 7-59
    - Control, 7-57
    - Flowchart, 7-55
    - State Diagram, 7-58
    - Timing, 7-56
  - Control Signals, 7-2
  - Controller, 1-2
  - Data, 5-3, 7-3, 7-32ff
  - Error,
    - Late, With DSACKx, Timing, 7-50
    - Exception, 8-4, 9-49
    - Signal, 5-6, 7-4, 7-21ff, 8-4, 8-15, 8-18
    - Without DSACKx Timing, 7-49
  - Errors, 7-47
  - Exceptions, 7-45
  - Fault Recovery, 8-18
  - Operation, 7-1
  - Synchronization, 7-53
  - Transfer Signals, 7-1
- Bus Grant, 7-57
  - Signal, 5-5, 7-55
- Bus Grant Acknowledge, 7-57
  - Signal, 5-6, 7-55
- Bus Request, 7-55
  - Signal, 5-5, 7-55
- Busy Primitive, 9-28
- Byte
  - Data Select, 7-20
  - Read Cycle,
    - Flowchart, 7-23
    - 32-Bit Port, Timing, 7-25
  - Write Cycle, 32-Bit Port, Timing, 7-30

— C —

- CA Bit, 9-27
- CAAR, 1-5, 2-4, 6-3
- Cache,
  - Filling, 7-19
  - Instruction, 1-9, 6-1, 6-3
  - Interactions, 7-19
  - Memory, 1-10
  - Organization, 6-1
  - Reset, 6-2
- Cache Address Register, 1-5, 2-4, 6-2
- Cache Control Register, 1-5, 2-4, 6-2
- Cache Disable Signal, 5-6, 6-2
- CACR, 1-5, 2-4
- Calculate Effective Address Timing Table, 10-15
- Calculate Immediate Effective Address Timing Table, 10-16
- Calculations, Execution Time, 10-7ff
- Capabilities, Addressing, 2-19
- CAS Instruction, 7-34
  - Example, 3-194
- Case,
  - Best, 10-5
  - Instruction Cache, 10-6
- CAS2 Instruction, 7-34
  - Example, 3-194
- CCR, 2-4, 3-12
- CDIS Signal, 5-6, 6-2
- Changing Privilege Level, 4-3
- Characteristics,
  - Electrical, 12-1
    - DC, 12-2
  - Thermal, 12-1
- Chip Select Coprocessor Signal, 11-6
- CIR, 9-7, 9-9, 9-21
  - Command, 9-24
  - Condition, 9-25
  - Control, 9-23
  - Instruction Address, 9-26
  - Operand, 9-25
  - Operand Address, 9-26
  - Operation Word, 9-24
  - Register Select, 9-26
  - Response, 9-23
  - Restore, 9-24
  - Save, 9-24
- Clear Entry in Instruction Cache Bit, 6-3
- Clear Instruction Cache Bit, 6-3
- CLK Signal, 5-7
- Clock Signal, 5-7
- Command CIR, 9-24
- Command Words, Illegal, Coprocessor Detected, 9-48
- Compare and Swap Instruction, 7-34
- Compatibility, M68000 Addressing, 2-27
- Computation, Condition Code, 3-200
- Concurrent Operation, 9-2
- Condition CIR, 9-25
- Condition Code
  - Computation, 3-200
  - Register, 2-4, 3-12

— C —

- Condition Tests, 3-201
- Condition Words, Illegal, Coprocessor Detected, 9-48
- Conditional Branch Instruction Timing Table, 10-36
- Connections, Power Supply, 5-7
- Control,
  - Bus Arbitration, 7-57
- Control CIR, 9-23
- Control Instruction Timing Table, 10-37
- Controller,
  - Bus, 1-2
  - Micro Bus, 1-2
- Coprocessor,
  - Communication Cycle, 7-60
  - Conditional Instructions, 9-10
  - Context Restore Instruction, 9-21
  - Context Save Instruction, 9-19
  - Data Processing Exceptions, 9-48
  - DMA, 9-4
  - Format Words, 9-17
  - General Instruction Protocol, 9-9
  - General Instructions, 9-8
  - Identification Code, 9-3
  - Instruction Format, 9-3
  - Instruction Summary, 9-54–9-57
  - Instructions, 3-10
  - Interface, 9-1, 9-23
  - MC68881, 9-1, 11-6
  - MC68882, 9-1, 11-6
  - Non-DMA, 9-4
  - Reset, 9-54
  - Response Primitive, 9-26
  - Response Primitive Format, 9-27
  - State Frames, 9-16
  - System Related Exceptions, 9-49
- Coprocessor Detected
  - Exceptions, 9-47
  - Format Errors, 9-49
  - Illegal Command Words, 9-48
  - Illegal Condition Words, 9-48
  - Protocol Violations, 9-49
- Coprocessor Interface Register, 9-6, 9-23
- Cp-ID, 7-62, 9-3
- cpBcc Instruction, 9-10
- cpDBcc Instruction, 9-13
- cpRESTORE Instruction, 8-22, 9-18
- cpSAVE Instruction, 8-22, 9-18
- cpScc Instruction, 9-12
- cpTRAPcc Instruction, 9-14
- cpTRAPcc Instruction Exception, 9-52
- CPU Space, 7-38, 9-5
- CPU Space Address Encoding, 7-38
- Cycle
  - Read, 7-23
  - Breakpoint Acknowledge, 7-40
  - Coprocessor Communication, 7-43
- Cycle (Continued)
  - Interrupt Acknowledge, 7-38
  - Interrupt Acknowledge, Autovector, 7-39
- Cycles, Data Transfer, 7-22

— D —

- Data, Immediate, 2-16
- Data Buffer Enable Signal, 5-4, 7-4, 7-23ff
- Data
  - Bus, 5-3, 7-3, 7-32ff
    - Requirements, Read Cycle, 7-7
    - Write Enable Signals, 7-19
  - Movement Instructions, 3-3
  - Port Organization, 7-6
  - Register Direct Mode, 2-8
  - Registers, 1-4, 2-1
  - Select, Byte, 7-20
  - Transfer
    - Cycles, 7-22
    - Transfer Mechanism, 7-5
  - Types, 1-5
- Data Strobe Signal, 5-4, 7-4, 7-22ff
- Data Transfer and Size Acknowledge Signals, 5-4, 7-4, 7-5, 7-20ff
- $\overline{\mathrm{DBEN}}$ Signal, 5-4, 7-4, 7-23ff
- DC Electrical Characteristics, 12-2
- Definitions, AC Electrical Specifications, 12-6
- Delay, Input, 7-1
- Description, General, 1-1
- Descriptions, Instruction, 3-16–3-193
- DFC, 1-5, 2-4
- Diagram, Block, 1-2
- Dimensions, Package, 13-2
- DMA Coprocessor, 9-4
- Double Bus Fault, 7-53, 8-5
- Doubly-Linked List
  - Deletion Example, 3-196
  - Insertion Example, 3-196
- $\overline{\mathrm{DR}}$ Bit, 9-28
- $\overline{\mathrm{DS}}$ Signal, 5-4, 7-4, 7-22ff
- $\overline{\mathrm{DSACK0}}$ Signal, 5-4, 6-8, 7-4, 7-5, 7-21ff
- $\overline{\mathrm{DSACK1}}$ Signal, 5-4, 6-8, 7-4, 7-5, 7-21ff
- Dynamic Bus Sizing, 7-5, 7-11, 7-21
- D31–D0 Signals, 5-3, 7-3, 7-34ff
- D7–D0, 1-3

— E —

- $\overline{\mathrm{ECS}}$ Signal, 5-3, 7-2, 7-21ff
- Effective Address Encoding Summary, 2-17
- E Bit, 6-3
- Electrical Characteristics, 12-1
  - DC, 12-2

— E —

- Electrical Specifications,
  - AC,
    - Clock Input, 12-3
    - Definitions, 12-6
    - Read/Write Cycles, 12-7
- Empty/Reset Format Word, 9-18
- Enable Cache Bit, 6-3
- Encoding,
  - Address Offset, 7-6
  - Size Signal, 7-6
- Errors, Bus, 7-47
- Evaluate and Transfer Effective Address Primitive, 9-33
- Evaluate Effective Address and Transfer Data Primitive, 9-33
- Example,
  - CAS Instruction, 3-194
  - CAS2 Instruction, 3-194
  - Doubly-Linked List
    - Deletion, 3-196
    - Insertion, 3-196
  - Linked List
    - Deletion, 3-195
    - Insertion, 3-195
- Exception,
  - Address Error, 8-5, 9-54
  - Breakpoint Instruction, 8-14
  - Bus Error, 8-4, 9-54
  - cpTRAPcc Instruction, 9-52
  - Format Error, 8-9
  - Illegal Instruction, 8-6
  - Instruction Trap, 8-6
  - Interrupt, 8-9, 9-53
  - Priority, 8-15
  - Privilege Violation, 8-7, 9-52
  - Processing, 4-4
    - Sequence, 8-1
    - State, 4-1
  - Reset, 8-2
  - Return from, 8-16
  - Stack Frame, 4-5, 8-22
  - Trace, 8-7, 9-52
  - Unimplemented Instruction, 8-6
  - Vector
    - Assignments, 8-2
    - Numbers, 8-1
  - Vectors, 4-5
- Exception Related
  - Instruction Timing Table, 10-38
- Exceptions,
  - Bus, 7-45
  - Coprocessor Data Processing, 9-48
  - Coprocessor Detected, 9-47
  - Coprocessor System Related, 9-49
  - F-Line Emulator, 8-7, 9-51
  - Multiple, 8-16
  - Primitive Processing, 9-47
- Execution Time Calculations, 10-7ff
- External Cycle Start Signal, 5-3, 7-2, 7-21ff

— F —

- F-Line, 9-3
  - Emulator Exceptions, 8-7, 9-51
- Fault, Double Bus, 7-53, 8-5
- FC2–FC0 Signals, 5-3, 6-2, 7-3, 7-34ff
- Fetch Effective Address Timing Table, 10-12
- Fetch Immediate Effective Address Timing Table, 10-13
- Floating Point Units, 11-6
- Flowchart,
  - Byte Read Cycle, 7-24
  - Long Word Read Cycle, 7-23
  - Read-Modify-Write Cycle, 7-35
  - Write Cycle, 7-28
  - Breakpoint Acknowledge, 7-43
  - Bus Arbitration, 7-55
  - Interrupt Acknowledge Cycle, 7-40
- Format,
  - Coprocessor Instruction, 9-3
  - Coprocessor Response Primitive, 9-27
  - Instruction, 3-1
  - Instruction Description, 3-16
- Format Error Exception, 8-9
- Format Errors,
  - Coprocessor Detected, 9-49
  - Main Processor Detected, 9-49
- Format Word,
  - Empty Reset, 9-18
  - Invalid, 9-18
  - Not Ready, 9-18
  - Valid, 9-19
- Format Words, Coprocessor, 9-17
- Formula, Instruction Cache Case Time, 10-1
- Freeze Cache Bit, 6-3
- Function Code Registers, 1-5, 2-4
- Function Code Signals, 5-3, 6-2, 7-3, 7-30ff

— G —

- General Description, 1-1
- Grant, Bus, 7-57
- Groups, Signal, 5-1

— H —

- Halt Operation, 7-50
  - Timing, 7-52
- Halt Signal, 5-6, 7-4, 7-22ff
- $\overline{\text{HALT}}$ Signal, 5-6, 7-4, 7-22ff
- Halted State, 4-1

— I —

Identification Code, Coprocessor, 9-3
Illegal Instruction Exception, 8-6
Immediate Data, 2-16
Index, Signal, 5-2
Indexed Addressing, 2-19, 2-20
Indirect Absolute Memory Addressing, 2-21
Indirect Addressing, 2-22, 2-23
Information, Ordering, 13-1
Initial Reset Timing, 7-60
Input Delay, 7-1
Instruction,
  BKPT, 7-40, 8-14
  Branch on Coprocessor Condition, 9-10
  Breakpoint, 7-40, 8-14
  CALLM, 11-1
  CAS, 7-34
  CAS2, 7-34
  Compare and Swap, 7-34
  Coprocessor Context Restore, 9-21
  Coprocessor Context Save, 9-19
  cpBcc, 9-10
  cpDBcc, 9-13
  cpRESTORE, 8-22, 9-18
  cpSAVE, 8-22, 9-18
  cpScc, 9-12
  cpTRAPcc, 9-14
  No Operation, 7-53
  NOP, 7-53
  RTE, 8-16
  RTM, 11-1
  Set on Coprocessor Condition, 9-12
  STOP, 8-9
  TAS, 7-34
  Test and Set, 7-34
  Test Coprocessor Condition, Decrement and Branch, 9-13
  Trap on Coprocessor Condition, 9-14
Instruction Address CIR, 9-26
Instruction Cache, 1-9, 6-1, 6-3, 10-1
    Time Formula, 10-8
Instruction Description
  Format, 3-16
  Notation, 3-2, 3-11
Instruction Descriptions, 3-16–3-193
Instruction Format, 3-1
  Summary, 3-201–3-217
Instruction Set, 1-6, 1-8
Instruction Timing Tables, 10-8
Instruction Trap Exception, 8-6
Instructions,
  Binary Coded Decimal, 3-8
  Bit Field, 3-7
  Bit Manipulation, 3-7
  Coprocessor, 3-10
    Conditional, 9-10
    General, 9-8
Instructions (Continued)
  Data Movement, 3-3
  Integer Arithmetic, 3-3
  Logical, 3-4
  Module, 11-1
  Multiprocessor, 3-10
  Privileged, 8-7
  Program Control, 3-8
  Rotate, 3-4
  Shift, 3-4
  System Control, 3-9
Integer Arithmetic Instructions, 3-3
Interface,
  Coprocessor, 9-1, 9-23
Internal Operand Representation, 7-6
Internal to External Data Bus Multiplexer, 7-7
Interrupt Acknowledge Cycle, 7-38
  Flowchart, 7-40
  Timing, 7-41
Interrupt
  Cycle, Spurious, 7-40
  Exception, 8-9, 9-53
  Levels, 8-10
Interrupt Pending Signal, 5-5, 8-11, 8-13
Interrupt Priority Level Signals, 5-5, 7-38, 8-10
Invalid Format Word, 9-18
$\overline{\text{IPEND}}$ Signal, 5-5, 8-11, 8-13
$\overline{\text{IPL0}}$–$\overline{\text{IPL2}}$ Signals, 5-5, 7-38, 8-10

— J —

Jump Effective Address Timing Table, 10-18

— L —

Late Bus Error,
  With $\overline{\text{DSACKx}}$, Timing, 7-50
Levels, Interrupt, 8-10
Linked List
  Deletion Example, 3-195
  Insertion Example, 3-195
Logical Instructions, 3-4
Long Word Read Cycle,
  Asynchronous, Flowchart, 7-23
  16-Bit Port, Timing, 7-27
  32-Bit Port, Timing, 7-27
  8-Bit Port, Timing, 7-26
Long Word to Long Word Transfer,
  Misaligned, 7-17
Long Word to Word Transfer, 7-8
  Misaligned, 7-17
Long Word Write Cycle,
  16-Bit Port, Timing, 7-32
  8-Bit Port, Timing, 7-31

— M —

- Maximum Ratings, 12-1
- MC68851 Signals, 11-6
- MC68881 Coprocessor, 9-1, 11-6
- MC68882 Coprocessor, 9-1, 11-6
- Mechanism, Data Transfer, 7-5
- Memory,
  - Cache, 1-10
  - Virtual, 1-6
- Memory Data Organization, 2-6
- Memory Indirect Postindexed Mode, 2-11
- Memory Indirect Preindexed Mode, 2-11
- Mid-Instruction Stack Frame, 9-44
- Misaligned
    - Long Word to Long Word Transfer, 7-19
    - Word to Long Word Transfer, 7-15
    - Word to Word Transfer, 7-17
  - Long Word to Long Word Transfer, 7-17
  - Long Word to Word Transfer, 7-13
    - Timing, 7-14
  - Operand, 7-11, 7-13
  - Word to Word Transfer, 7-15
  - Word to Word Transfer Timing, 7-16
- Mode,
  - Absolute
    - Long Address, 2-16
    - Short Address, 2-16
  - Address Registers
    - Direct, 2-8
    - Indirect, 2-8
    - Indirect Displacement, 2-9
    - Indirect Index (Base Displacement), 2-10
    - Indirect Index (8-Bit Displacement), 2-10
    - Indirect Postincrement, 2-8
    - Indirect Predecrement, 2-9
  - Data Register Direct, 2-8
  - Memory Indirect
    - Postindexed, 2-11
    - Preindexed, 2-11
  - Program Counter
    - Indirect Displacement, 2-12
    - Indirect Index (Base Displacement), 2-13
    - Indirect Index (8-Bit Displacement), 2-12
    - Memory Indirect Postindexed, 2-14
    - Memory Indirect Preindexed, 2-15
- Model, Programming, 1-3, 9-4
- Modes, Addressing, 1-5, 2-5
- Module, 11-1
  - Call, 11-5
  - Description, 11-1
  - Return, 11-5
  - Stack Frame, 11-3
- MOVE Instruction,
  - Special Purpose, Timing Table, 10-28
  - Timing Table, 10-28
- Multiple Exceptions, 8-15
- Multiplexer, Data Bus, Internal to External, 7-9
- Multiprocessor Instructions, 3-10
- M68000 Family, 2-27

— N —

- Nested Subroutine Calls, 3-197
- No Operation Instruction, 7-53
- Non-DMA Coprocessor, 9-4
- NOP Instruction, 7-53
- Normal Processing State, 4-1
- Not Ready Format Word, 9-18
- Notation, Instruction Description, 3-2, 3-11
- Null Primitive, 9-29

— O —

- $\overline{\text{OCS}}$ Signal, 5-3, 7-3, 7-22ff
- Operand, Misaligned, 7-11, 7-13
- Operand Address CIR, 9-26
- Operand CIR, 9-25
- Operand Cycle Start Signal, 5-3, 7-3, 7-22ff
- Operands, 2-1
- Operation,
  - Concurrent, 9-2
  - Halt, 7-50
  - Reset, 7-60
  - Retry, 7-48
- Operation Word CIR, 9-24
- Operations, Bit Field, 3-198
- Ordering Information, 13-1
- Organization,
  - Cache, 6-1
  - Data Port, 7-6
  - Memory Data, 2-4
  - Register Data, 2-1
- Overlap, 10-5

— P —

- Package Dimensions, 13-2
- PC Bit, 9-28
- Pin Assignment, 13-2
- Pipeline, 1-10
- Pipeline Synchronization, 3-199
- Pipelined Architecture, 1-9
- Post-Instruction Stack Frame, 9-46
- Power Considerations, 12-1
- Power Supply Connections, 5-7
- Pre-Instruction Stack Frame, 9-45
- Primitive,
  - Busy, 9-28
  - Coprocessor Response, 9-26
  - Evaluate and Transfer Effective Address, 9-33
  - Evaluate Effective Address and Transfer Data, 9-33

— P —

Primitive (Continued)
  Null, 9-29
  Supervisor Check, 9-30
  Take Address and Transfer Data, 9-37
  Take Mid-Instruction Exception, 9-44
  Take Post-Instruction Exception, 9-45
  Take Pre-Instruction Exception, 9-43
  Transfer from Instruction Stream, 9-32
  Transfer Main Processor Control Register, 9-38
  Transfer Multiple Coprocessor Registers, 9-40
  Transfer Multiple Main Processor Registers, 9-39
  Transfer Operation Word, 9-31
  Transfer Single Main Processor Register, 9-38
  Transfer Status Register and ScanPC, 9-41
  Transfer to from Top of Stack, 9-37
  Write to Previously Evaluated Effective Address, 9-35
Primitive Processing Exception, 9-47
Priority, Exception, 8-15
Privilege Level,
  Changing, 4-3
  Supervisor, 4-1, 4-2
  User, 4-1, 4-3
Privilege Violation Exception, 8-7, 9-52
Privileged Instructions, 8-7
Processing, Exception, 4-4
Processor Activity,
  Even Alignment, 10-4
  Odd Alignment, 10-4
Processor Generated Reset Timing, 7-61
Program Control Instructions, 3-8
Program Counter
  Indirect Displacement Mode, 2-12
  Indirect Index (Base Displacement) Mode, 2-13
  Indirect Index (8-Bit Displacement) Mode, 2-12
  Memory Indirect Postindexed Mode, 2-14
  Memory Indirect Preindexed Mode, 2-15
Programming Model, 1-3
Protocol
  Processor General Instruction, 9-9
  Violations,
    Coprocessor Detected, 9-50
    Main Processor Detected, 9-50

— Q —

Queue, 2-30

— R —

R W Signal, 5-4, 7-3, 7-23ff
Ratings, Maximum, 12-1
Read Cycle,
  32-Bit Port, Timing, 7-29
  Data Bus Requirements, 7-7

— R —

Read-Modify-Write
  Cycle, 7-36
    Byte, 32-Bit Port, Timing, 7-36
    Flowchart, 7-35
  Signal, 5-4, 7-3, 7-34ff
Read Write Signal, 5-4, 7-3
Recovery,
  Bus Fault, 8-19
  RTE, 8-21
  Software, 8-21
Register,
  Cache Address, 1-5, 2-4, 6-3
  Cache Control, 1-4, 2-4, 6-2
  Condition Code, 2-4, 3-12
  Coprocessor Interface, 9-6, 9-23
  Status, 1-4, 2-4, 8-8, 8-10, 8-14
  Vector Base, 1-5, 2-4
  Data Organization, 2-1
Register Select CIR, 9-26
Registers,
  Address, 1-3, 2-3
  Data, 1-3, 2-1
  Function Code, 1-5, 2-4
Representation, Internal Operand, 7-6
Request, Bus, 7-55
Requirements, Data Bus, Read Cycle, 7-7
Reset,
  Cache, 6-2
  Coprocessor, 9-54
  Exception, 8-2
  Operation, 7-60
  Signal, 5-6, 7-79ff
RESET Signal, 5-6, 7-55
Response CIR, 9-23
Restore CIR, 9-24
Retry Operation, 7-48
  Late,
    Asynchronous, Timing, 7-51
Return from Exception, 8-16
RMC Signal, 5-4, 7-3, 7-36ff
Rotate Instructions, 3-4
RTE
  Bus Fault Recovery, 8-21
  Instruction, 8-17

— S —

Save CIR, 9-24
ScanPC, 9-12
Sequence, Exception Processing, 8-1
Sequencer, 1-10
Set, Instruction, 1-6, 1-8
Set on Coprocessor Condition Instruction, 9-12
SFC, 1-5, 2-4
Shift Instructions, 3-4

— S —

- Signal,
  - Address Strobe, 5-4, 7-2, 7-3, 7-22ff
  - AS, 5-4, 7-2, 7-3, 7-22ff
  - Autovector, 5-5, 7-5, 8-14
  - AVEC, 5-5, 7-5, 8-14
  - BERR, 5-6, 7-5, 7-22ff, 8-4, 8-15, 8-18
  - BG, 7-55, 9-5
  - BGACK, 5-6, 7-55
  - BR, 5-5, 7-34
  - Bus Error, 5-6, 7-4, 7-21ff, 8-4, 8-15, 8-18
  - Bus Grant, 5-5, 7-55
  - Bus Grant Acknowledge, 5-6, 7-57
  - Bus Request, 5-5, 7-55
  - Cache Disable, 5-6, 6-2
  - CDIS, 5-6, 6-2
  - CLK, 5-7
  - Clock, 5-7
  - Data Buffer Enable, 5-4, 7-4, 7-23ff
  - Data Strobe, 5-4, 7-4, 7-22ff
  - DBEN, 5-4, 7-4, 7-23ff
  - DS, 5-4, 7-4, 7-22ff
  - DSACK0, 5-4, 7-4, 7-5, 7-21ff
  - DSACK1, 5-4, 7-4, 7-5, 7-21ff
  - ECS, 5-3, 7-2, 7-21ff
  - External Cycle Start, 5-3, 7-2, 7-22ff
  - Halt, 5-6, 7-4, 7-22ff
  - HALT, 5-6, 7-4, 7-22ff
  - Interrupt Pending, 5-5, 8-11, 8-13
  - IPEND, 5-5, 8-11, 8-13
  - OCS, 5-3, 7-3, 7-22ff
  - Operand Cycle Start, 5-3, 7-3, 7-22ff
  - R W, 5-4, 7-3, 7-23ff
  - Read-Modify-Write, 5-4, 7-3, 7-23ff
  - Read Write, 5-4, 7-3, 7-23ff
  - Reset, 5-6, 7-55
  - RESET, 5-6, 7-55
  - RMC, 5-4, 7-3, 7-36ff
  - SIZ0, 5-3, 7-3, 7-7, 7-19, 7-23ff
  - SIZ1, 5-3, 7-3, 7-7, 7-19, 7-23ff
- Signal Assertion Results, Asynchronous Cycle, 7-47
- Signal Groups, 5-1
- Signal Index, 5-2
- Signal Summary, 5-7
- Signals,
  - A1–A0, 7-6, 7-7, 7-17ff
  - A31–A0, 5-3, 7-3
  - Bus Control, 7-2
  - Bus Transfer, 7-1
  - Data Bus Write Enable, 7-19
  - Data Transfer and Size Acknowledge, 5-4, 7-4, 7-5, 7-20ff
  - D31–D0, 5-3, 7-3, 7-34ff
  - FC2–FC0, 5-3, 6-2, 7-3, 7-22ff
  - Function Code, 5-3, 6-2, 7-3, 7-22ff
  - Interrupt Priority Level, 5-5, 7-38, 8-10
  - IPL2–IPL0, 5-5, 7-38, 8-10
  - MC68851, 11-6
- Single Operand Instruction Timing Table, 10-32
- Size Signal Encoding, 7-7
- Sizing, Dynamic Bus, 7-5, 7-13, 7-20
- SIZ0 Signal, 5-3, 7-3, 7-7, 7-19, 7-23ff
- SIZ1 Signal, 5-3, 7-3, 7-7, 7-19, 7-23ff
- Software Bus Fault Recovery, 8-20
- Space, CPU, 7-38, 9-5
- Special Status Word, 8-19
- Spurious Interrupt Cycle, 7-43
- SR, 1-5, 2-4, 8-8, 8-10, 8-14
- Stack,
  - System, 2-29
  - User Program, 2-29
- Stack Frame,
  - Exception, 4-5, 8-23
  - Mid-Instruction, 9-44
  - Post-Instruction, 9-46
  - Pre-Instruction, 9-45
- State,
  - Diagram, Bus Arbitration, 7-58
  - Exception Processing, 4-1
  - Halted, 4-1
  - Normal Processing, 4-1
- State Frames, Coprocessor, 9-16
- Status Register, 1-4, 2-4, 8-8, 8-10, 8-14
- Status Word, Special, 8-19
- STOP Instruction, 8-9
- Structure Addressing, 2-22
- Subroutine Calls, Nested, 3-197
- Summary,
  - Addressing Mode, 2-23
  - Coprocessor Instruction, 9-54–9-57
  - Effective Address Encoding, 2-17
  - Instruction Format, 3-201–3-219
  - Signal, 5-7
- Supervisor Check Primitive, 9-30
- Supervisor
  - Privilege Level, 4-1, 4-2
- Synchronization,
  - Bus, 7-53
  - Pipeline, 3-199
- Synchronous
  - Sample Window, 7-2
- System
  - Control Instructions, 3-9
  - Stack, 2-29

— T —

- Table
  - Timing, 10-9
- Tables, Instruction Timing, 10-8
- Take Address and Transfer Data Primitive, 9-37
- Take Mid-Instruction Exception Primitive, 9-44
- Take Post-Instruction Exception Primitive, 9-45
- Take Pre-Instruction Exception Primitive, 9-43
- TAS Instruction, 7-36

—T—

- Test and Set Instruction, 7-36
- Test Coprocessor Condition, Decrement and Branch Instruction, 9-13
- Tests, Condition, 3-201
- Thermal Characteristics, 12-1
- Timing,
  - Asynchronous
    - Byte Read Cycle, 32-Bit Port, 7-25
    - Byte Read-Modify-Write Cycle, 32-Bit Port, 7-36
    - Byte Write Cycle, 32-Bit Port, 7-30
    - Read Cycle, 32-Bit Port, 7-29
    - Word Read Cycle, 32-Bit Port, 7-25
    - Word Write Cycle, 32-Bit Port, 7-30
    - Write Cycle, 32-Bit Port, 7-29
  - Autovector Interrupt Acknowledge Cycle, 7-39
  - Breakpoint Acknowledge Cycle, 7-40
    - Exception Signaled, 7-45
  - Bus Arbitration, 7-54
    - Bus Inactive, 7-59
  - Bus Error,
    - Late, With $\overline{\text{DSACKx}}$, 7-50
    - Without $\overline{\text{DSACKx}}$, 7-49
  - Bus Synchronization, 7-53
  - Halt Operation, 7-52
  - Initial Reset, 7-60
  - Interrupt Acknowledge Cycle, 7-38
  - Long Word,
    - Read Cycle, 16-Bit Port, 7-27
    - Read Cycle, 32-Bit Port, 7-27
    - Read Cycle, 8-Bit Port, 7-26
    - Write, 7-10
    - Write Cycle, 16-Bit Port, 7-32
    - Write Cycle, 8-Bit Port, 7-31
  - Misaligned
    - Long Word to Word Transfer, 7-14
    - Word to Word Transfer, 7-16
  - Processor Generated Reset, 7-61
  - Retry Operation, Late, 7-51
  - Write, Long Word, 7-10
  - Write, Word, 7-12
- Timing Diagrams, AC Electrical, 12-7
- Timing Table,
  - Arithmetic Logical Instruction, 10-29
    - Immediate, 10-30
  - Binary Coded Decimal Instruction, 10-31
  - Bit Field Instruction, 10-35
  - Bit Manipulation Instruction, 10-34
  - Calculate Effective Address, 10-15
  - Calculate Immediate Effective Address, 10-16
  - Conditional Branch Instruction, 10-36
  - Control Instruction, 10-37
  - Exception Related
    - Instruction, 10-38
  - Fetch Effective Address, 10-12
  - Fetch Immediate Effective Address, 10-13
  - Jump Effective Address, 10-18
  - MOVE Instruction, 10-19
    - Special Purpose, 10-28
  - Restore Operation, 10-39
  - Save Operation, 10-39
  - Shift Rotate Instruction, 10-33
  - Single Operand Instruction, 10-32
- Trace Exception, 8-8, 9-52
- Transfer,
  - Long Word to Long Word, Misaligned, 7-19
  - Long Word to Word, 7-9
  - Misaligned
    - Word to Long Word, 7-15
    - Word to Word, 7-17
    - Long Word to Long Word, 7-17
    - Long Word to Word, 7-13
    - Word to Word, 7-15
    - Word to Word, Timing, 7-16
  - Word to Byte, 7-11
- Transfer from Instruction Stream Primitive, 9-32
- Transfer Main Processor Control Register Primitive, 9-38
- Transfer Multiple Coprocessor Registers Primitive, 9-40
- Transfer Multiple Main Processor Registers Primitive, 9-39
- Transfer Operation Word Primitive, 9-31
- Transfer Single Main Processor Register Primitive, 9-38
- Transfer Size Signals, 5-3, 7-3, 7-6, 7-19, 7-30ff
- Transfer Status Register and ScanPC Primitive, 9-41
- Transfer to from Top of Stack Primitive, 9-37
- Trap on Coprocessor Condition Instruction, 9-14
- Types,
  - Address Space, 4-4
  - Data, 1-5

—U—

- Unimplemented Instruction Exception, 8-6
- Units, Floating Point, 11-6
- User Privilege Level, 4-1, 4-3
- User Program Stack, 2-29

—V—

- Valid Format Word, 9-19
- VBR, 1-5, 2-4
- V_{CC} Pin Assignments, 13-2
- Vector
  - Base Register, 1-5, 2-4
  - Numbers, Exception, 8-1
- Vectors, Exception, 4-5
- Virtual Machine, 1-6
- Virtual Memory, 1-6

I

— W —

Window,
  Asynchronous Sample, 7-2
Word, Special Status, 8-19
Word Read Cycle, Asynchronous, 32-Bit Port,
  Timing, 7-28
Word to Byte Transfer, 7-11
Word to Long Word Transfer, Misaligned 7-15
Word to Word Transfer, Misaligned, 7-17
Word Write Cycle, Asynchronous, 32-Bit Port,
  Timing, 7-33
Word Write Timing, 7-12
Write Cycle,
  Asynchronous, 7-28
    Flowchart, 7-28
    32-Bit Port, Timing, 7-29
Write Timing,
  Long Word, 7-10
  Word, 7-12
Write to Previously Evaluated Effective Address
  Primitive, 9-35

1 Introduction

2 Data Organization and Addressing Capabilities

3 Instruction Set

4 Processing States

5 Signal Description

6 On-Chip Cache Memory

7 Bus Operation

8 Exception Processing

9 Coprocessor Interface Description

10 Instruction Execution Timing

11 Applications Information

12 Electrical Specifications

13 Ordering Information and Mechanical Data

I Index

PRENTICE HALL, Englewood Cliffs, N.J. 07632

ISBN 0-13-567017-